www.clubedelphi.net Feita para desenvolvedores

ClubeDelphi

Ano 6 · Edição 75 · R$8,90

DevMedia GROUP

ISSN 1517990-7

Aplicações Multi Banco

Adapte sua aplicação para funcionar com diferentes BDs, usando técnicas de desenvolvimento em camadas e POO

Interagindo com o Sistema Operacional

Veja como manipular informações do SO através de funções avançadas

Dicas de Delphi

Confira diversas dicas nas áreas de VCL, banco de dados e WEB

Coluna: Ask the Expert!

Mini-Curso: Construindo uma aplicação Web Completa Parte II

Implementando cadastros, consultas e customização de controles

POO no Delphi Parte II

Como implementar orientação a objetos em aplicações Delphi

Especial - AJAX com Delphi

Faça sua aplicação entrar na WEB 2.0!

Comprando esta edição tenha acesso na WEB a:

+130 artigos e +140 vídeo-aulas

Proteja seu Software

ANTIPIRATARIA

DISTRIBUIÇÃO SEGURA

AUMENTO DOS SEUS LUCROS

PROTEÇÃO DA
PROPRIEDADE INTELECTUAL

Compact500®

Segurança. Qualidade. Eficiência.

Isto é a solução Compact-500. Você tem a garantia de proteção do seu software contra pirataria e engenharia reversa, além de controle total do gerenciamento de seu programa. Tudo isto de maneira fácil e extremamente eficiente.

Só uma empresa com um histórico de inovação, bom atendimento e sucesso pode garantir a segurança do seu bem mais valioso: suas informações.

Compact-500
A segurança do seu software

SafeNet®
The Foundation of Information Security

São Paulo (11) 3392 4600 www.safenet-inc.com/brasil

ClubeDelphi

Ano 6 - 75ª Edição - 2006 - ISSN 1517990-7

Impresso no Brasil

Editorial

Você já parou para pensar qual a melhor forma de aliar as poderosas técnicas de programação orientada a objetos (POO), para criar uma camada independente de acesso a banco de dados, abstraindo a aplicação de detalhes específicos do SGBD e tipo de conectividade? Nesta edição, César, Miguel e Francisco trazem na matéria de capa uma solução para alcançar esses objetivos. Conheça as diferenças existentes na linguagem SQL para cada tipo de BD, como Firebird, Oracle, SQL Server e PostgreSQL. Saiba os cuidados necessários para não tornar sua aplicação dependente de um único BD, principalmente se o seu cliente puder optar por outro fornecedor. Veja ainda como usar a POO para criar classes para fazer a persistência e abstração do acesso a BD.

Ultimamente, temos ouvido falar muito em tecnologias como AJAX. Não tenho dúvida que estamos diante de uma grande revolução, pois aquilo que parecia ser a pior limitação da Web, está com os dias contados. Com o AJAX, podemos limitar os postbacks ao servidor, evitando refreshs totais da página. Podemos chamar métodos no servidor, de forma assíncrona e sem atualizar a página, e então ajustar pequenas porções da tela. Será que as aplicações Web vão por fim de uma vez por todas, à era Desktop/Win32? Não quero arriscar uma opinião prematura, mas confesso que é muito importante estar preparado para isso. No artigo do Fabrício, saiba como utilizar AJAX em suas aplicações Delphi com ASP.NET.

Confira ainda muito sobre POO, programação com API, o mini-curso do Luciano Pimenta que mostra a criação de uma loja virtual completa usando Delphi e ASP.NET e muito mais! Uma ótima leitura, sucesso com o Delphi e um grande abraço.

Guinther Pauli – guinther@devmedia.com.br
Editor Geral

NÃO PERCA

PORTAL DO ASSINANTE

A ClubeDelphi tem uma novidade para você que comprou este exemplar na banca de jornal: você pode acessar GRATUITAMENTE, o Portal do Assinante ClubeDelphi!

Confira o que você encontra no Portal do Assinante:

- Mais de 130 Vídeo Aulas!
- Diversos Mini-Cursos Gratuitos!
- 1 Livro Eletrônico sobre ADO.NET e BDP!
- Mais de 130 Artigos Exclusivos!

Para Utilizar o Portal do Assinante, acesse www.devmedia.com.br/clubedelphi/portal.asp e utilize as informações abaixo:

Login: DVM.CD
Senha: X1LMZ

O acesso é válido por 30 dias a partida da data de lançamento da revista. Todos os meses a ClubeDelphi lhe dará uma senha válida para acessar o portal. Comprando a revista regularmente em bancas, você terá acesso ininterrupto a ele!

Corpo Editorial

Diretor Editorial e Administrativo
Gladstone Matos
gladstone@neoficio.com.br

Designer
Vinicius O. Andrade
viniciusoandrade@gmail.com

Capa
Antonio Xavier
jaxs_s_design@yahoo.com.br

Revisão
Fabio Correa
fabiocorrea@devmedia.com.br

Articulistas desta edição
Everson Volaco, Rodolfo de Faria, Cesar Blumm, Miguel R. Fornari, Francisco M. Trindade, Fabricio Desbessel, João Carlos da Silva, Alessandréia de Oliveira, Marco Araújo.

Na Web
www.devmedia.com.br/clubedelphi/portal.asp

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Luciano Pimenta
lucianopimenta@clubedelphi.net

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Publicidade
Para informações sobre veiculação de anúncio na revista ou no site entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

Luciano Pimenta - Editor do Site
lucianopimenta@clubedelphi.net

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Thiago Andrade – Atendimento ao Leitor
www.devmedia.com.br/central/default.asp
(21) 2220-5375

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 2220-5375

Parcerias

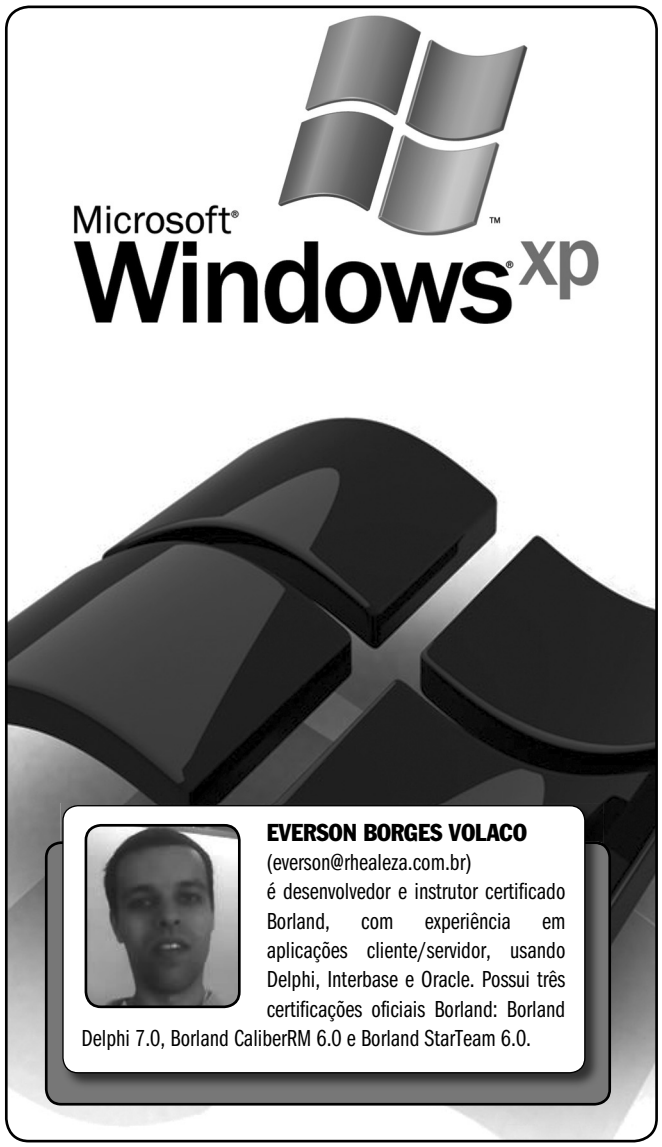
Para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Jeff Wendell
jeff@devmedia.com.br

ASSESPRO PARANÁ

API do Windows

Programação com Funções Nativas do SO



EVERSON BORGES VOLACO
(everson@rhealeza.com.br)
é desenvolvedor e instrutor certificado Borland, com experiência em aplicações cliente/servidor, usando Delphi, Interbase e Oracle. Possui três certificações oficiais Borland: Borland Delphi 7.0, Borland CaliberRM 6.0 e Borland StarTeam 6.0.

Neste artigo utilizaremos alguns dos vários métodos disponíveis na API do Windows, para capturar as mais diversas informações do sistema operacional e das aplicações que nele são executadas. Através da API do Windows podemos realizar as mais diversas tarefas, como: acessar informações de hardware, software, interagir com outros programas, criar objetos, alterar configurações e comportamentos do sistema.

> **Nota:** Para o exemplo deste artigo utilizei o Delphi 7 Enterprise e o Windows XP Professional. Dependendo do seu sistema operacional, uma ou outra função do exemplo terá que ser adaptada.

Criando a aplicação de exemplo

Abra o Delphi 7 e inicie uma nova aplicação. Altere o nome do formulário para "FrmPrincipal" e salve a unit como "untFrmPrincipal.pas". Para o arquivo de projeto dê o nome de "API_Windows.dpr". Adicione alguns componentes visuais e configure-os de acordo com a **Figura 1**.

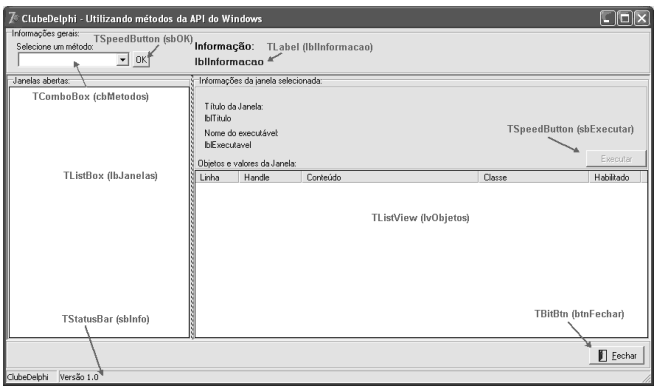


Figura 1. Layout da aplicação de exemplo em tempo de design

A idéia do exemplo é disponibilizar no *cbMetodos* uma lista pré-definida de opções onde cada uma utilizará métodos da API do Windows para retornar a informação solicitada. Concentraremos vários desses métodos em uma *unit* separada do formulário, a fim de facilitar seu uso em outros projetos.

Terminado o desenho da tela, vamos iniciar a implementação do código. Selecione o *cbMetodos* e digite a seguinte lista de opções dentro da sua propriedade *Items*:

```
<Selecione>
Usuário
Máquina
IP
Sistema Operacional
Processador
Clock
Memória
Janelas abertas
```

Ainda com o componente selecionado, altere sua propriedade *ItemIndex* para "0". Dessa forma o primeiro item da lista ficará selecionado por padrão. No evento *OnChange* do *cbMetodos* digite o seguinte código:

```
lblInformacao.Caption := '';
lblTitulo.Caption := '';
lblExecutavel.Caption := '';
lbJanelas.Items.Clear;
lvObjetos.Items.Clear;
sbExecutar.Enabled := False;
```

Utilizamos o código anterior para "limpar" as informações dos componentes da tela quando o usuário selecionar uma opção da lista. Antes de implementarmos o botão OK, vamos criar as funções que serão chamadas a partir das opções disponíveis na lista. Crie uma nova *unit* (*File>New>Unit*) e salve-a como "untFuncoes.pas". Logo abaixo da seção *interface* da nova unit, adicione a cláusula *uses* e coloque a referência às seguintes units:

```
uses Windows, Dialogs, Classes, SysUtils, Winsock,
  TLHelp32, ComCtrls, Registry, ComObj, Forms;
```

Utilizaremos classes e métodos dessas units em nossas funções de acesso a API do Windows. Muitos dos métodos da API estão mapeados dentro da unit *Windows*. Antes da seção *implementation* declare as funções que serão implementadas dentro da unit, conforme a **Listagem 1**.

Listagem 1. Funções para a utilização do exemplo

```
{ Retorna o IP da máquina corrente }
function RetornarIP: string;
{ Retorna o nome da máquina corrente }
function RetornaNomeDaMaquina: string;
{ Retorna o login do usuário logado }
function RetornarUsuarioLogado: string;
{ Retorna o nome do arquivo executável de um aplicativo }
function RetornarNomeExe(const pHandle: hwnd): string;
{ Retorna o nome do processador da máquina corrente }
function RetornarProcessador: string;
{ Retorna a versão do sistema operacional da máquina corrente }
function RetornarSO: string;
{ Retorna o clock da máquina corrente }
function RetornarClock: string;
{ Retorna o total de memória da máquina corrente }
function RetornarMemoria: string;
```

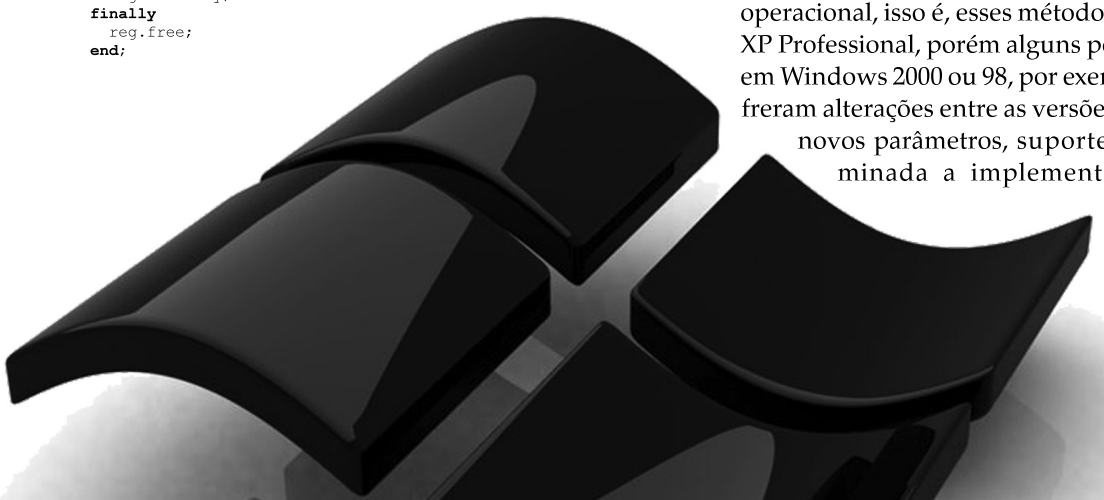
Antes da declaração de cada função há uma breve explicação (na forma de comentário) sobre sua finalidade. Veremos que algumas dessas funções chegam a utilizar código *assembler* para conseguir capturar a informação que necessita. Não nos preocuparemos muito com o "como funciona" cada linha das funções, pois o objetivo é disponibilizar uma forma de obter tal informação apenas. Veja na **Listagem 2** o código-fonte que implementa as funções recém-declaradas.

Listagem 2. Implementação dos métodos na unit untFuncoes.pas

```
function RetornarIP: string;
var
  WSAData: TWSAData;
  HostEnt: PHostEnt;
  Name : String;
begin
  try
    WSAStartup(2, WSAData);
    SetLength(Name, 255);
    Gethostname(PChar(Name), 255);
    SetLength(Name, StrLen(PChar(Name)));
    HostEnt := gethostbyname(PChar(Name));
    with HostEnt^ do
      Result := Format('%d.%d.%d.%d',
        [Byte(h_addr^[0]),Byte(h_addr^[1]),
        Byte(h_addr^[2]),Byte(h_addr^[3])]);
    WSACleanup;
  except
    on E: Exception do
      MessageDlg('Erro ao tentar capturar o IP' + #13 +
        'Mensagem original: ' + E.Message, mtError, [mbOK], 0);
  end;
end;

function RetornaNomeDaMaquina: string;
const
  Buff_Size = MAX_COMPUTERNAME_LENGTH + 1;
var
  lpBuffer : PChar;
  nSize : DWord;
begin
  Result := '';
  try
    nSize := Buff_Size;
    lpBuffer := StrAlloc(Buff_Size);
    GetComputerName(lpBuffer,nSize);
    Result := AnsiUpperCase(string(lpBuffer));
    StrDispose(lpBuffer);
  except
    on E: Exception do
      MessageDlg('Erro ao tentar capturar' + #13 +
        ' o nome da máquina' + #13 +
        'Mensagem original: ' + E.Message, mtError, [mbOK], 0);
  end;
end;

function RetornarUsuarioLogado: string;
var
  lpUserName : PAnsiChar;
  lpnLength  : DWORD;
begin
  try
    lpnLength := 0;
    WNetGetUser (nil, nil, lpnLength);
    if lpnLength > 0 then
    begin
      GetMem (lpUserName, lpnLength);
      if WNetGetUser (nil, lpUserName,
        lpnLength) = NO_ERROR then
        Result := AnsiUpperCase(lpUserName)
      else
        Result := '';
      FreeMem (lpUserName, lpnLength);
    end
    else
      Result := '';
  except
    on E : Exception do
      MessageDlg('Erro ao tentar capturar' + #13 +
        ' o usuário logado' + #13 +
        'Mensagem original: ' + E.Message, mtError, [mbOK], 0);
  end;
end;
```

```
function RetornarNomeExe(const pHandle: hwnd): string;
var
 vPID: DWord;
 vASnapShotHandle: THandle;
 vContinueLoop: Boolean;
 vAProcessEntry32: TProcessEntry32;
begin
  try
    GetWindowThreadProcessID(pHandle, @vPID);
    vASnapShotHandle := CreateToolHelp32SnapShot(TH32CS_SNAPPROCESS, 0);
    vAProcessEntry32.dwSize := SizeOf(vAProcessEntry32);
    vContinueLoop := Process32First(vASnapShotHandle,
    { Aponta o ponteiro para o inicio do Task Manager}
    vAProcessEntry32);
    { Varre a lista de tasks do Task Manager }
    while Integer(vContinueLoop) <> 0 do
    begin
      { Caso encontre o pid então }
      if vAProcessEntry32.th32ProcessID = vPID then
      begin
           {Retorna o nome do executável utilizado por este pid}
        result:= vAProcessEntry32.szExeFile;
        break;
      end;
      { Continua o loop (next) }
      vContinueLoop := Process32Next(vASnapShotHandle,
        vAProcessEntry32);
    end;
      CloseHandle(vASnapShotHandle);
  except
    on E : Exception do
      MessageDlg('Erro ao tentar capturar ' + #13 +
        'o nome do executável' + #13 +
        'Mensagem original: ' + E.Message, mtError, [mbOK], 0);
  end;
end;

function RetornarProcessador: string;
var
  reg: TRegIniFile;
begin
  try
    reg := TRegIniFile.create;
    try
      reg.RootKey := HKEY_LOCAL_MACHINE;
      reg.OpenKey('HARDWARE\DESCRIPTION\SYSTEM\'+
        'CentralProcessor\', false);
      Result := reg.ReadString('0', 'Identifier', '') +
        ' ' + reg.ReadString('0', 'ProcessorNameString', '');
    finally
      reg.free;
    end;
  except
    on E: Exception do
      MessageDlg('Erro ao tentar capturar ' + #13 +
        'o nome do processador' + #13 +
        'Mensagem original: ' + E.Message, mtError, [mbOK], 0);
  end;
end;

function RetornarSO: string;
var
  reg: TRegIniFile;
begin
  try
    reg := TRegIniFile.create;
    try
      reg.RootKey := HKEY_LOCAL_MACHINE;
      reg.OpenKey('SOFTWARE\Microsoft\Windows NT\', false);
     if (reg.ReadString('currentversion', 'ProductName', '') = '') then
        Result := 'WindowsNT ' + reg.ReadString(
          'currentversion', 'CurrentVersion', '')
      else
        Result := reg.ReadString('currentversion',
          'ProductName', '');
      reg.CloseKey;
    finally
      reg.free;
    end;
  except
    on E: Exception do
      MessageDlg('Erro ao tentar capturar a ' + #13 +
        ' versão do sistema operacional' + #13 +
        'Mensagem original: ' + E.Message, mtError, [mbOK], 0);
  end;
end;

function RetornarClock: string;
const
  DelayTime = 500;
var
  TimerHi, TimerLo: DWORD;
  PriorityClass, Priority: Integer;
  cpuspeed: string;
begin
  try
    PriorityClass := GetPriorityClass(GetCurrentProcess);
    Priority := GetThreadPriority(GetCurrentThread);
    SetPriorityClass(GetCurrentProcess, REALTIME_PRIORITY_CLASS);
    SetThreadPriority(GetCurrentThread, THREAD_PRIORITY_TIME_CRITICAL);
    Sleep(10);
    asm
          dw 310Fh // rdtsc
          mov TimerLo, eax
          mov TimerHi, edx
    end;
    Sleep(DelayTime);
    asm
          dw 310Fh // rdtsc
          sub eax, TimerLo
          sbb edx, TimerHi
          mov TimerLo, eax
          mov TimerHi, edx
    end;
    SetThreadPriority(GetCurrentThread, Priority);
    SetPriorityClass(GetCurrentProcess, PriorityClass);
    cpuspeed := FormatFloat('0.00', (
      TimerLo / (1000.0 * DelayTime)));
    Result := cpuspeed;
  except
    on E: Exception do
      MessageDlg('Erro ao tentar ' + #13 +
        ' capturar o clock' + #13 +
        'Mensagem original: ' + E.Message, mtError,
        [mbOK], 0);
  end;
end;

function RetornarMemoria: string;
var
  MemoryStatus: TMemoryStatus;
begin
  try
    MemoryStatus.dwLength := sizeof(MemoryStatus);
    GlobalMemoryStatus(MemoryStatus);
    Result := FormatFloat('0.00',
      MemoryStatus.dwTotalPhys / 1000000);
  except
    on E : Exception do
      MessageDlg('Erro ao tentar capturar ' + #13 +
        ' o total de memória ' + #13 +
        'Mensagem original: ' + E.Message, mtError, [mbOK], 0);
  end;
end;
```

Navegando pelo código fonte dessas funções percebemos que as mesmas utilizam tipos, métodos e comandos que não são comuns no dia a dia do desenvolvedor. A API do Windows possui métodos em C, os quais não possuem um padrão 100% compatível entre as versões do sistema operacional, isso é, esses métodos funcionam no Windows XP Professional, porém alguns podem deixar de funcionar em Windows 2000 ou 98, por exemplo. Alguns métodos sofreram alterações entre as versões do Windows, ganhando novos parâmetros, suporte a novos tipos etc. Terminada a implementação da unit, volte ao formulário e adicione a *untFuncoes* a seção *uses* através da opção de menu *File>Use Unit*.



Listando as janelas ativas do sistema operacional

Voltando ao *cbMetodos*, repare que adicionamos um item chamado *Janelas abertas* em sua propriedade *Items*. A idéia é que, quando essa opção for selecionada, sejam listadas todas as aplicações abertas no Windows dentro do *lbJanelas* (*ListBox*). Ao selecionar uma janela no *lbJanelas* será carregada a lista de objetos dessa dentro do *lvObjetos* (*ListView*).

Para que possamos implementar essa funcionalidade, vamos adicionar mais alguns métodos da API, porém agora dentro da *unit* do nosso formulário. Antes da implementação dos métodos, declare as seguintes variáveis na seção *var* da *unit*:

```
var
  ValorTexto: Array [1..255] of Char;
  Conteudo: string;
  Linha: Integer;
  handleJanela: HWND;
  PidPrograma: Cardinal;
```

Essas variáveis estão em um escopo global dentro da *unit*, isso é, não faz parte da classe *TFrmPrincipal*. Utilizaremos essas variáveis dentro das funções do Windows que manipularemos a seguir. Para que possamos listar todas as janelas ativas adicione a função da **Listagem 3** dentro da seção *implementation*.

Listagem 3. Função para listar todas as janelas ativas

```
function EnumWindowsProc(
  Wnd: HWND; lb : TListbox): BOOL; stdcall;
var
  caption : Array [0..128] of Char;
begin
  Result := True;
  if IsWindowVisible(Wnd) and ((GetWindowLong(Wnd, GWL_HWNDPARENT) = 0) or
    (HWND(GetWindowLong(Wnd, GWL_HWNDPARENT)) = GetDesktopWindow)) and
    ((GetWindowLong(Wnd, GWL_EXSTYLE) and WS_EX_TOOLWINDOW) = 0) then
  begin
    SendMessage( Wnd, WM_GETTEXT, Sizeof(caption),
      integer(@caption));
    lb.Items.AddObject( caption,TObject( Wnd ));
  end;
end;
```

Utilizaremos a função anterior em conjunto com a função *EnumWindows* (API do Windows) para capturar o título de todas as janelas abertas em nosso sistema operacional, técnica muito utilizada em softwares que precisam monitorar aplicações executadas por usuários em uma máquina. Para que possamos testar as funções já implementadas vamos adicionar o código da **Listagem 4** no evento *OnClick* do botão *sbOK*

Listagem 4. Código do sbOK

```
lblInformacao.Caption := '';
lbJanelas.Items.Clear;
case cbMetodos.ItemIndex of
  0: MessageDlg('Selecione um método da lista', mtInformation, [mbOk], 0);
  1: lblInformacao.Caption := RetornarUsuarioLogado;
  2: lblInformacao.Caption := RetornaNomeDaMaquina;
  3: lblInformacao.Caption := RetornarIP;
  4: lblInformacao.Caption := RetornarSO;
  5: lblInformacao.Caption := RetornarProcessador;
  6: lblInformacao.Caption := RetornarClock;
  7: lblInformacao.Caption := RetornarMemoria;
  8: EnumWindows(@EnumWindowsProc, integer(lbJanelas));
end;
```

No código da listagem anterior, verificamos o índice do item selecionado no *cbMetodos* e executamos o método equivalente à opção. Com exceção do índice 8, todos os demais têm a informação retornada pela função armazenada dentro do *lblInformacao*.

No caso da opção de índice 8 (*Janelas abertas*) utilizamos o método *EnumWindows* para realizar um *loop* em todas as janelas abertas e através do método *EnumWindowsProc* fazemos a captura do texto da barra de título de cada janela e adicionamos ao *lbJanelas*.

Nesse ponto você já pode compilar e rodar a aplicação. Veja a aplicação em execução na **Figura 2**.

Listando os objetos de uma janela ativa

Dando continuidade ao exemplo, vamos implementar agora a funcionalidade para listar todos os objetos de uma janela além de possibilitar ao usuário uma pequena interação com os objetos dessa janela.

Dentro da seção *Informações da janela selecionada* iremos trazer o título da janela e o nome do executável que possui tal janela, além de várias informações sobre os objetos presentes dentro da mesma. No formulário, adicione as funções da **Listagem 5**, dentro da seção *implementation*:

Através dos métodos *EnumProcess* e *EnumChildProc* podemos varrer uma determinada janela a partir do seu *handle* e acessar todos os objetos presentes dentro dela. Para que possamos testar esses métodos selecione o *lbJanelas* e digite o código da **Listagem 6** para o seu evento *OnClick*.

Quando o usuário clicar em um item do *ListBox*, fazemos a captura do *handle* da janela selecionada através do texto da barra de título da mesma e passamos o *handle* como parâmetro para o método *EnumProcess*, que por sua vez chama o método *EnumChildProc*, que varre todos os objetos e adiciona diversas informações sobre eles no *lvObjetos*.

Utilizamos ainda o *RetornarNomeExe* da unit *untFuncoes*, para capturar o nome do executável que contém a janela em questão. Uma vez listados os objetos da janela, vamos adicionar uma funcionalidade para interagirmos com esses objetos, desde que os mesmos sejam do tipo *Button*. Selecione o *lvObjetos* e digite o seguinte código ao seu evento *OnSelectItem*:

```
if (lvObjetos.Selected <> nil) then
  if (AnsiLowerCase(lvObjetos.Selected.SubItems[2]) = 'button') then
    sbExecutar.Enabled := True
  else
    sbExecutar.Enabled := False;
```

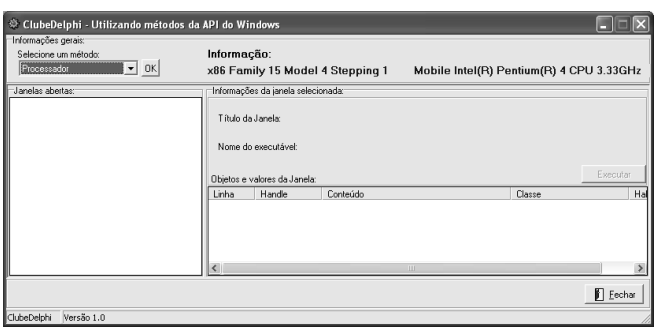


Figura 2. Buscando informações do processador utilizado na máquina

Listagem 5. Métodos para captura de informações de uma janela ativa e de seus objetos

```
function EnumChildProc(hHwnd: HWND;
  vX: Integer): Boolean; stdcall;
var
  nomeClasse, Title, caption, habilitado: string;
  passwordChar: Integer;
  flag: Boolean;
begin
  Result := False;
  try
     if (hHwnd = NULL) then
         result := false
     else
     begin
         flag := False;
         passwordChar := 0;
         if SendMessage(hHwnd, EM_GETPASSWORDCHAR, 0, 0) <> 0 then
         begin
           flag := True;
           passwordChar := SendMessage(hHwnd, EM_GETPASSWORDCHAR, 0, 0);
           PostMessage(hHwnd, EM_SETPASSWORDCHAR, 0, 0);
           Sleep (500);
           SetForegroundWindow(hHwnd);
         end;
         SendMessage( hHwnd, WM_GETTEXT, Sizeof(ValorTexto),
           integer(@ValorTexto));
         Conteudo := strpas(@ValorTexto);
         setLength(Conteudo,length(conteudo));

         if flag then
         begin
           PostMessage(hHwnd, EM_SETPASSWORDCHAR,
             passwordChar, 0);
           SetForegroundWindow(hHwnd);
         end;

         SetLength(nomeClasse, 255);
         SetLength(nomeClasse,GetClassName(hHwnd,
           PChar(nomeClasse),Length(nomeClasse)));
         SetLength(title, 255);
         SetLength(title, GetWindowText(hHwnd,
           PChar(title), Length(title)));

         Linha  := Linha + 1;

         SetLength(caption, 255);
         SetLength(caption, GetWindowText(
           handleJanela, PChar(caption), Length(caption)));
         if (AnsiUpperCase(caption) = AnsiUpperCase(
           FrmPrincipal.lbJanelas.Items[
           FrmPrincipal.lbJanelas.ItemIndex])) then
         begin
           if IsWindowEnabled(hHwnd) then
             habilitado := 'SIM'
           else
             habilitado := 'NÃO';
           with FrmPrincipal.lvObjetos.Items.Add do
           begin
             Caption := IntToStr(Linha); //Linha
             SubItems.Add(IntToStr(hHwnd)); //Handle
             SubItems.Add(conteudo); //Valor(Conteudo)
             SubItems.Add(nomeClasse);
             SubItems.Add(habilitado);
           end;
         end;
         Result := true;
     end;
  except
    on E: Exception do
      MessageDlg(E.Message, mtError, [mbOk], 0);
  end;
end;

function EnumProcess(hHwnd: HWND;
  lParam: integer): boolean; stdcall;
var
  pPid, LdCld: DWORD;
  title: string;
begin
  Result := False;
  try
    if (hHwnd = NULL) then
      Result := False
    else
    begin
      pPid := 0;
      LdCld := 0;
      GetWindowThreadProcessId(hHwnd, pPid);
      SetLength(title, 255);
      SetLength(title, GetWindowText(hHwnd,
        PChar(title), Length(title)));
      handleJanela := hHwnd;
      PidPrograma := pPid;
      Linha  := 0;
      EnumChildWindows(hHwnd,@EnumChildProc,
        LdCld);
    end;
    Result := true;
  except
    on E: Exception do
      MessageDlg(E.Message, mtError, [mbOk], 0);
  end;
end;
```

Listagem 6. Evento OnClick do lbJanelas

```
procedure TFrmPrincipal.lbJanelasClick(
  Sender: TObject);
var
  handle: HWND;
begin
  if lbJanelas.Items[
    lbJanelas.ItemIndex] <> '' then
  begin
    lvObjetos.Clear;
    handle := FindWindow(nil, PChar(
      lbJanelas.Items[lbJanelas.ItemIndex]));
    if handle = 0 then
    begin
      MessageDlg('Janela "' + lbJanelas.Items[
        lbJanelas.ItemIndex] + '" não encontrada!',
        mtInformation, [mbOk], 0);
      Exit;
    end;
    Screen.Cursor := crSQLWait;
    EnumProcess(handle, 0);
    lblTitulo.Caption := lbJanelas.Items[
      lbJanelas.ItemIndex];
    lblExecutavel.Caption := RetornarNomeExe(handleJanela);
    Screen.Cursor := crDefault;
  end;
end;
```

No código anterior, verificamos se o usuário selecionou algum item e se a coluna *Classe* possui o valor *Button*, indicando que o objeto é do tipo botão. Dependendo do tipo do objeto selecionado habilitamos ou não o botão *sbExecutar* de nossa aplicação.

A idéia do *sbExecutar* é permitir o envio de um *click* para o botão da janela externa selecionada. Para finalizar nosso exemplo selecione o *sbExecutar* e digite o seguinte código em seu evento *OnClick*:

```
SetForegroundWindow(handleJanela);
SendMessage(handleJanela, WM_COMMAND, MAKEWPARAM(GetWindowLong(StrToInt(
  lvObjetos.Selected.SubItems[0]), GWL_ID),
  BN_CLICKED), StrToInt(lvObjetos.Selected.SubItems[0]));
```

No código anterior, utilizamos o *SetForegroundWindow* para colocar a janela selecionada em primeiro plano. Através do método *SendMessage*, enviamos uma simulação do comando *click* para o botão selecionado na lista de objetos da janela através do *handle* da janela e do *handle* do botão. Compile novamente a aplicação e rode a mesma para que possamos testá-la.

Testando a aplicação de exemplo

Para que possamos testar a aplicação, execute a mesma e abra diversos aplicativos no Windows. Entre esses aplicativos abra a Calculadora que acompanha o sistema operacional. Selecione uma opção qualquer no *cbMetodos* e clique sobre o botão OK para capturar a informação desejada.

Caso seja selecionada a última opção, todas as janelas abertas serão listadas no *ListBox*. Selecione o item referente à janela Calculadora. Repare que todos os objetos presentes são listados no *ListView*.

Figura 3. "Manipulando" a calculadora do Windows a partir de uma aplicação Delphi

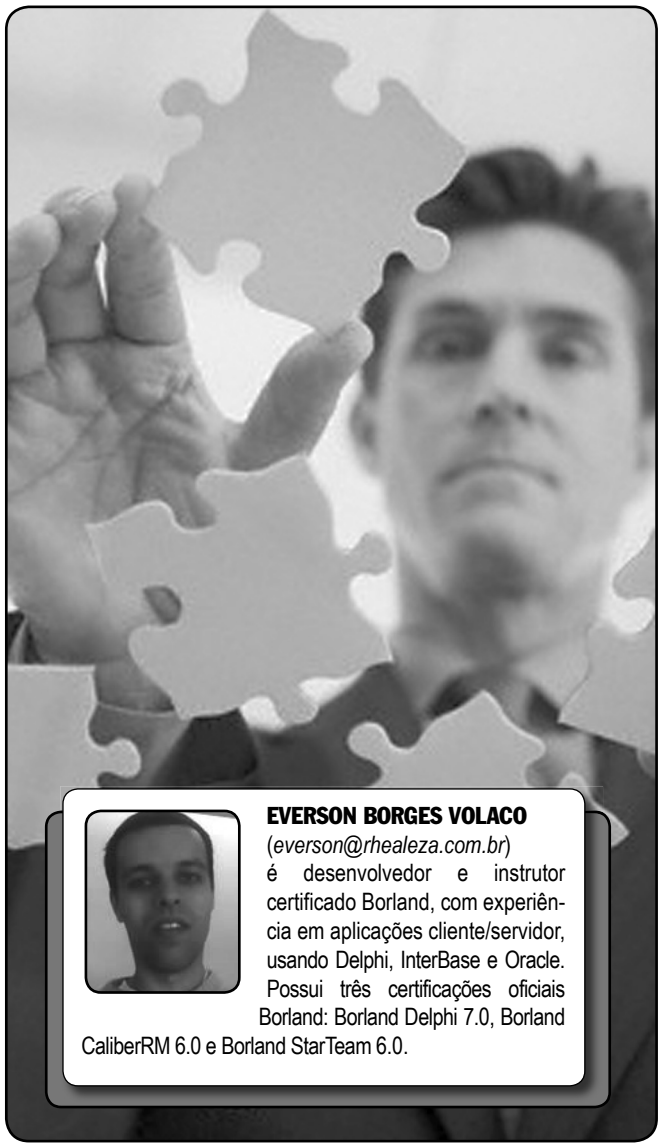
Vamos utilizar nossa aplicação para acessar os botões da calculadora e efetuar alguns cálculos. Selecione os objetos que possuem sua classe definida como *Button* e clique no botão *Executar* para enviar o comando. Veja a aplicação interagindo com a calculadora do Windows na **Figura 3**.

Conclusão

Vimos neste artigo que através da API do Windows, podemos ter acesso a uma infinidade de opções do sistema operacional, além interagir com aplicativos externos e até mesmo criar nossos próprios objetos e janelas. Meu objetivo foi mostrar que através de aplicativos Delphi, podemos obter e manipular as mais diversas informações do SO, por mais específicas que elas possam ser. Um abraço e até a próxima. ■

15 Dicas de Delphi

Dicas de Delphi para Desenvolvedores Iniciantes



EVERSON BORGES VOLACO
(*everson@rhealeza.com.br*)
é desenvolvedor e instrutor certificado Borland, com experiência em aplicações cliente/servidor, usando Delphi, InterBase e Oracle. Possui três certificações oficiais Borland: Borland Delphi 7.0, Borland CaliberRM 6.0 e Borland StarTeam 6.0.

Este artigo mostrará 15 dicas de Delphi (Win32) para desenvolvedores iniciantes e intermediários. Veremos os mais variados assuntos, desde o IDE até opções para facilitar a depuração de aplicações. Usuários que estão iniciando com o Delphi ou que possuem pouca experiência no mesmo, terão algumas dicas úteis para uso no dia a dia durante o desenvolvimento.

Nota: Todas as dicas são focadas no Delphi 7, porém, a maioria pode ser aplicada em qualquer versão do Delphi em aplicações Win32.

1. Definindo o carregamento dos formulários da aplicação

Quando criamos uma aplicação no Delphi, todos os formulários que construímos a partir da opção *New Form* por padrão ficam definidos como *Auto-create forms* dentro da janela *Options* do projeto (*Project>Options>Forms*).

Isso significa que, se você tiver, por exemplo, 10 formulários em sua aplicação, todos serão carregados em memória no momento que a aplicação for iniciada. Dependendo do tamanho de sua aplicação você pode ter sérios problemas de performance.

Uma boa prática é definir o formulário principal da aplicação como *Auto-create forms* e os demais como *Available forms* (**Figura 1**).

Uma observação apenas é que se o formulário a ser chamado estiver definido dentro da seção *Auto-create forms*, não precisamos criar a mesmo via código, isso é, basta fazer a chamada ao método *Show* (ou *ShowModal*) para visualizar o formulário.

Já se o formulário estiver definido como *Available forms* precisaremos criar o mesmo através de seu construtor (método *Create*) antes de invocar o método *Show* para mostrá-lo na tela.

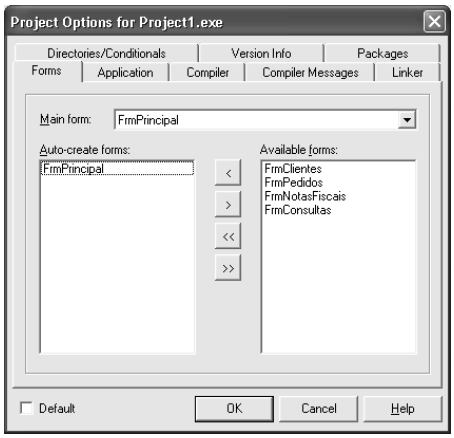


Figura 1. Definindo o comportamento de criação dos formulários do projeto

Nota: Você pode especificar a seção para que um formulário seja adicionado por padrão quando for criado (opção *New Form*) durante o projeto. Para isso, acesse a opção *Tools>Environment Options>Designer* e defina a opção *Auto create forms & data modules* disponível no item *Module creation options*.

2. Criando e destruindo formulários

Quando instanciamos um formulário em nossa aplicação através do seu construtor (método *Create*) e fazemos sua chamada através do método *ShowModal* para mostrar o mesmo ao usuário da aplicação, é importante destruirmos sua instancia quando o usuário fechar o mesmo.

Uma boa opção para garantir que o formulário sempre será destruído, mesmo que algum erro ocorra, é utilizar a instrução *try...finally...end*. Essa instrução garante que todo o código escrito dentro da seção *finally* será executado mesmo que algum erro grave ocorra na aplicação. Veja o exemplo na **Listagem 1.**

Listagem 1. Usando o bloco try...finally...end para criação de um formulário

```
FrmClientes := TFrmClientes.Create(Self);
try
  FrmClientes.ShowModal;
finally
  FrmClientes.Release;
  FrmClientes := nil;
End;
```

No código da listagem anterior, criamos o *FrmClientes* e dentro do bloco *try* realizamos a chamada ao mesmo. Como estamos utilizando o método *ShowModal*, a aplicação só continuará a execução do código acima após o usuário ter fechado o *FrmClientes*. Quando essa operação ocorrer, o código do bloco *finally* será executado, realizando a destruição da instância do *FrmClientes*.

Nota: A instrução *try...finally...end* não funciona com o método *Show*, pois o mesmo não "trava" a aplicação até o formulário ser fechado. Nesse caso, o código do bloco *finally* será executado logo após a chamada ao método *Show*, não permanecendo assim o formulário disponível para o usuário realizar suas operações.

3. Utilizando Data Modules

Outra boa prática é fazer o uso de Data Modules na aplicação para separar os componentes de acesso a dados da interface com o usuário (formulários). Utilizando Data Modules, podemos separar as regras de negócio que fazem acesso ao banco de dados de nossos formulários além de podermos reutilizar os *DataSets* em diferentes formulários.

Dependendo do tamanho de sua aplicação é aconselhável ainda criar vários Data Modules, cada um contendo os *DataSets* referente a um módulo ou segmento da aplicação. Normalmente criam-se um Data Module principal (por exemplo, *DMPrincipal*) o qual contém o componente de conexão com o banco de dados além de um ou dois *DataSets* auxiliares.

Nota: *DataSets* auxiliares não possuem nenhuma instrução SQL pré-definida; os mesmos são utilizados por funções distribuídas na aplicação para realizarem operações no banco de dados através de SQLs passados em tempo de execução.

São criados então, "n" outros Data Modules, contendo cada um diversos *DataSets*, sendo todos ligados a conexão disponível no Data Module principal. Na **Figura 2** temos um exemplo de como seria a criação dos Data Modules no carregamento da aplicação.

Nota: Neste exemplo, é necessário fazer a chamado ao *Create* do Data Module, para ter acesso aos seus métodos e componentes.

4. Habilitando opções básicas do IDE

Quando instalamos o Delphi algumas opções básicas e interessantes do IDE não vêem marcadas por padrão. Você pode estar definindo essas opções através do menu

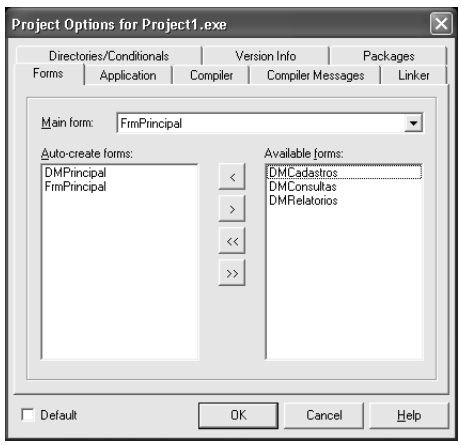


Figura 2. Utilizando Data Modules para o armazenamento dos componentes de acesso a dados da aplicação

Tools>Environment Options e *Tools>Debugger Options*.

Por exemplo, para que possamos visualizar a caixa de informações referente a compilação do projeto, mensagens de *warnings*, *hints* e *errors* marque a opção *Show compiler progress* disponível na aba *Preferences* da janela *Environment Options*.

Outra opção bastante útil que também não vem marcada por padrão é a *Show component captions* a qual, quando marcada passa, a mostrar o nome dos componentes não visuais dentro do formulário ou Data Module em tempo de design.

5. Passando instruções SQL em tempo de execução

Haverá situações onde precisaremos montar e enviar para o banco de dados instruções SQL de acordo com opções selecionadas pelo usuário na aplicação em tempo de execução. Componentes como *Query*, *SQLDataSet* e *ClientDataSet* possuem uma propriedade denominada *Params* a qual permite especificarmos parâmetros para filtros dentro da instrução SQL.

Porém, há situações onde toda a instrução SQL precisa ser montada em tempo de execução, tornando o uso da propriedade *Params* bastante complicada para ser utilizada. Uma saída é utilizar o caractere dois pontos (:) para definir os parâmetros via código ou atribuir os valores para os filtros diretamente dentro da instrução SQL.

Veja na **Listagem 2** dois exemplos de utilização de parâmetros em instruções SQL atribuídas em tempo de execução.

Listagem 2. Atribuindo parâmetros em tempo de execução

```
SQLDataSet1.Close;
SQLDataSet1.CommandText := 'SELECT CODIGO, NOME, '+
  'FONE FROM CLIENTES WHERE CODIGO = ' + Edit1.Text;
SQLDataSet1.Open;
ou
SQLDataSet1.Close;
SQLDataSet1.CommandText := 'SELECT CODIGO, NOME, '+
  'FONE FROM CLIENTES WHERE CODIGO = :COD';
SQLDataSet1.Params[0].AsInteger := StrToInt(Edit1.Text);
SQLDataSet1.Open;
```

Em ambas as instruções SQL, passamos o valor do *Edit1* como filtro para o campo CODIGO da tabela CLIENTES. Na primeira instrução, fazemos a atribuição do valor diretamente ao campo. Repare que apesar do campo CODIGO ser do tipo numérico (*integer*) não fazemos a conversão do valor armazenado na propriedade *Text* do *Edit1* para *Integer*.

Nesses casos, devemos passar o valor numérico como *string* dentro do código Delphi. Já no segundo caso, utilizamos o caractere dois pontos (:) para definir um parâmetro de nome COD. Terminada a montagem da instrução SQL utilizamos a propriedade *Params* para atribuir o valor do *Edit1* ao parâmetro.

Para esse caso, utilizamos a propriedade *AsInteger* e fazemos a conversão de tipo da propriedade *Text* do *Edit1* de *string* para *integer* (*StrToInt*). Para o primeiro caso, dependendo do tipo do campo que será utilizado no filtro dentro da instrução SQL, devemos passar o valor para o filtro de maneiras diferentes. Veja exemplos na **Tabela 1.**

Tipo	Valor	Exemplo
Integer	100	'WHERE CODIGO = ' + '100';
String	Cliente	'WHERE TIPO = uotedStr('Cliente');
Date	01/01/2006	'WHERE DTVENDA = ' + QuotedStr('01/01/2006')

Tabela 1. Exemplos de passagem de parâmetros

Para campos do tipo *string* ou *Date* você precisa passar o valor do filtro entre duas aspas simples, sendo isso que faz internamente o método *QuotedStr*. Você pode utilizar ainda, para o mesmo propósito, #39 ou ''''. Veja os exemplos a seguir:

```
'SELECT * FROM FORNECEDORES WHERE UF = #39 + 'PR' + #39;
'SELECT * FROM FORNECEDORES WHERE CIDADE = ' + ''''' +
  'Curitiba' + ''''';
```

6. Passando novas instruções SQL através do ClientDataSet

Quando utilizamos o trio de componentes *SQLDataSet*, *DataSetProvider* e *ClientDataSet* para fazer acesso ao banco de dados, definimos a instrução SQL diretamente na propriedade *CommandText* do *SQLDataSet*. Podem haver situações onde podemos ter a necessidade de alterar a instrução SQL do *SQLDataSet* em tempo de execução. Entretanto, dependendo da arquitetura da aplicação essa alteração pode ser bastante complicada.

Imagine em uma aplicação três camadas onde o *SQLDataSet* fica na aplicação servidora e o *ClientDataSet* fica na aplicação cliente. Para esses casos o *DataSetProvider* possui uma opção que habilita enviarmos uma nova instrução SQL a partir da propriedade *CommandText* do *ClientDataSet*.

Essa propriedade chama-se *poAllowCommandText* e está disponível dentro da propriedade *Options*. Modificando para *True*, podemos modificar a instrução SQL definida em tempo de design no *SQLDataSet* através do *ClientDataSet* em tempo de execução.

7. Manipulando arquivos texto com TStringList

Quem ainda não teve a necessidade de trabalhar com arquivos textos no Delphi mais cedo ou mais tarde vai acabar se deparando com essa necessidade. O Delphi traz o tipo de dado *TextFile* o qual permite manipularmos arquivos texto a partir de aplicações Delphi.

Você pode, porém, manipular arquivos texto de uma forma mais simples através do uso da classe *TStringList*. A classe permite criar e manipular em memória uma lista de *strings*, como por exemplo, um arquivo texto.

Através do método *LoadFromFile* podemos carregar um arquivo texto para dentro de uma variável do tipo *TStringList*. Através do índice de cada item (ou linha) da lista, podemos acessar e/ou alterar o conteúdo desse item.

Realizada a manipulação no arquivo texto, podemos utilizar o método *SaveToFile* para salvar novamente o arquivo no sistema operacional. Veja na **Listagem 3** um exemplo da utilização dessa técnica.

Listagem 3. Trabalhando com arquivos textos no Delphi

```
procedure TFrmPrincipal.btnCarregarArquivoClick(
  Sender: TObject);
var
  { Variável que recebe o conteúdo do arquivo texto }
  arquivo: TStringList;
  i: Integer;
begin
  { Instancia a variável arquivo }
  arquivo := TStringList.Create;
  try
    { Carrega o conteúdo do arquivo texto para a
      memória }
    arquivo.LoadFromFile('c:\temp\arquivo1.txt');
    { Realiza um loop em toda a lista }
    for i := 0 to arquivo.Count - 1 do
    begin
      { Mostra o valor atual da linha }
      ShowMessage('O conteúdo original da linha ' +
        IntToStr(i) + ' é ' + arquivo[i]);
      { Atribui um novo valor para a linha corrente }
      arquivo[i] := 'Novo conteúdo da linha: ' +
        IntToStr(i);
    end;
    { Salva as alterações no arquivo }
    arquivo.SaveToFile('c:\temp\arquivo1.txt');
  finally
    { Libera a instancia da lista da memória }
    FreeAndNil(arquivo);
  end;
end;
```

8. Utilizando MultiSelect no DBGrid

Sem dúvida um dos componentes mais utilizados no desenvolvimento de aplicações com Delphi é o *DBGrid*. Veremos nessa dica como utilizar a opção de *MultiSelect* do componente. Habilitando essa opção, o usuário poderá selecionar mais de um registro (ao mesmo tempo) dentro do grid.

Adicione um *DBGrid* e altere a propriedade *Options> dgMultiSelect* para *True*. Acesse um banco de dados qualquer e mostre no *DBGrid* registros de alguma tabela desse banco. Adicione um botão ao formulário e digite o código da **Listagem 4** em seu evento *OnClick*.

Listagem 4. MultiSelect no DBGrid

```
procedure TFrmPrincipal.btnMostrarSelecionadosClick(
  Sender: TObject);
var
  i: Integer;
  aux: string;
begin
  for i := 0 to DBGrid1.SelectedRows.Count - 1 do
  begin
    ClientDataSet1.GotoBookmark(pointer(
      DBGrid1.SelectedRows.Items[i]));
    aux := aux + IntToStr(ClientDataSet1.RecNo) +
     ' - ' +ClientDataSet1.FieldByName(
     'CUSTOMER').AsString + #13;
  end;
  ShowMessage('Linhas selecionadas: ' + #13 + aux);
end;
```

No código da listagem anterior utilizamos a propriedade *SelectedRows* do *DBGrid* para varrer todos os registros selecionados pelo usuário. Utilizamos o método *GotoBookmark* do *ClientDataSet* para posicionar o cursor no registro corrente selecionado. Veja na **Figura 3** a aplicação de exemplo em execução.

9. Utilizando as propriedades FetchOnDemand e PacketRecord do ClientDataSet

Uma das principais características do *ClientDataSet* é trabalhar com os dados em memória, desconectado do banco de dados. Quando trazemos uma grande quantidade de registros do banco de dados para serem mostrados em um *DBGrid* por exemplo, o *ClientDataSet* primeiro carrega os registros em memória para só depois mostrá-los na tela para o usuário.

Figura 3. Trabalhando com multiseleção de registros em um DBGrid

Dependendo do número de registros, o usuário pode demorar a visualizar os mesmos dentro do *DBGrid*. Isso ocorre porque, por padrão, a propriedade *PacketRecord* do *ClientDataSet* vem definida com o valor "–1". Através desta propriedade podemos controlar o número de registros por pacote que serão buscados no servidor de banco de dados.

Essa propriedade está diretamente vinculada a propriedade *FetchOnDemand*, que também por padrão vem definida como *True*. Você pode diminuir o tempo para visualização de grandes quantidades de registros na tela alterando o valor da propriedade *PacketRecord*. Faça a seguinte simulação: crie uma aplicação para mostrar todos os registros de uma tabela grande do banco de dados em um *DBGrid* utilizando *ClientDataSet*.

Você verá que os dados demorarão a aparecer no *DBGrid* devido ao fato do *ClientDataSet* ter que carregar os registros antes em memória. Volte a aplicação em tempo de design e altere a propriedade *PacketRecords* para "10". Rode novamente a aplicação. Os registros aparecerão instantaneamente no *DBGrid*.

O que ocorreu na verdade é que apenas os 10 primeiros registros foram carregados no *ClientDataSet*. A medida que navegamos pelos registros no *DBGrid* os demais registros vão sendo buscados no banco de dados, sempre em pacotes de 10 registros. Você pode ainda, se preferir, trazer apenas os 10 primeiros registros e através de uma opção na aplicação ir buscando os demais registros somente quando necessário.

Para isso, basta alterar a propriedade *FetchOnDemand* para *False* e utilizar o método *GetNextPacket* para buscar o próximo pacote de registros do servidor de banco de dados.

> **Nota:** Essas propriedades são muito interessantes em aplicações multicamadas, possibilitando assim um melhor controle dos *ResultSets* retornados pela aplicação servidora e passando ao usuário final uma sensação de melhor performance da aplicação.

10. Propriedade UniDirectional do componente TQuery (BDE)

Esaa dica é destinada aos desenvolvedores que utilizam BDE em conjunto com os componentes *DataSetProvider* e *ClientDataSet*. Programadores que utilizam o Delphi 5, versão a qual não possui a tecnologia dbExpress, para a criação de aplicações multicamadas ou o conjunto *Query* + *DataSetProvider* + *ClientDataSet*, precisam tomar cuidado com a duplicidade dos registros no buffer.

O *Query* possui uma propriedade denominada *UniDirectional*, a qual por padrão é definida como *False*. Dentro desse cenário como a tecnologia BDE não é unidirecional por natureza, como o dbExpress por exemplo, podemos ter duplicidade dos registros em cache, visto que tanto o *Query* como o *ClientDataSet* armazenarão os registros retornados pelo servidor de banco de dados.

Essa característica pode degradar a performance da aplicação, e o complicado é que nenhuma mensagem de erro é retornada para o desenvolvedor. Para esse tipo de aplicação é aconselhável a alteração da propriedade *UniDirectional* para *True*, fazendo assim com que o cache dos registros seja realizado apenas pelo *ClientDataSet*.

11. Atualizando informações da aplicação

É comum utilizarmos componentes como o *ProgressBar* e o *Gauge* para manter o usuário da aplicação informado sobre o progresso de um determinado processo dentro de nossa aplicação. Porém, dependendo do processo a aplicação pode travar sendo definida como *não respondendo* para o sistema operacional.

Para atualizar as informações da tela durante a execução de um processo pesado e/ou demorado de nossa aplicação podemos utilizar o método *ProcessMessages* da variável *Application*. Veja o exemplo na **Listagem 5**.

Listagem 5. Usando ProcessMessages para não travar o sistema

```
procedure TFrmPrincipal.btnExecutarClick(
  Sender: TObject);
var
  i: Integer;
begin
  ProgressBar1.Min := 0;
  ProgressBar1.Position := ProgressBar1.Min;
  ProgressBar1.Max := 100000;
  for i := 1 to 100000 do
  begin
    ProgressBar1.StepIt;
    Label1.Caption := IntToStr(i);
    Application.ProcessMessages;
  end;
end;
```

No código da listagem anterior, fazemos um loop de 1 até 100000 atualizando a cada passagem do loop um *ProgressBar* e um *Label*. Caso não utilizarmos o método *ProcessMessages* o *Label* não será atualizado na tela em tempo real para o usuário da aplicação.

12. Controlando a versão da sua aplicação

Podemos definir e controlar o número da versão de nossa aplicação Delphi. Dentro da janela de opções do projeto (*Project>Options*) na aba *Version Info* basta habilitar a opção *Include version information in project* e informar o número da versão corrente de sua aplicação dentro da seção *Module version number*.

Além do número da versão podemos entrar com diversas outras informações como nome da empresa, nome do produto, descrição, entre outras. O Delphi permite ainda que você crie novas chaves/valores dentro da seção *Key/Value* através da opção *Add Key* no menu de contexto.

Todas as informações são armazenadas no arquivo executável gerado pela nossa aplicação quando compilamos a mesma. Para acessar essas informações a partir do Windows, basta selecionar o arquivo executável e clicar sobre a opção *Propriedades* disponível no menu de contexto.

Através do número da versão do aplicativo podemos controlar atualizações e identificar a versão do aplicativo que está rodando em cada cliente. Você pode capturar esse número de versão em tempo de execução para mostrar ao usuário em uma tela do tipo *Sobre* ou *About*, por exemplo. Para isso, utilize o método da **Listagem 6**.

Listagem 6. Pegando a versão do arquivo

```
function GetVersaoArq: string;
var
  VerInfoSize: DWORD;
  VerInfo: Pointer;
  VerValueSize: DWORD;
  VerValue: PVSFixedFileInfo;
  Dummy: DWORD;
begin
  VerInfoSize := GetFileVersionInfoSize(PChar(
    ParamStr(0)), Dummy);
  GetMem(VerInfo, VerInfoSize);
  GetFileVersionInfo(PChar(ParamStr(0)), 0,
    VerInfoSize, VerInfo);
  VerQueryValue(VerInfo, '\', Pointer(VerValue),
    VerValueSize);
  with VerValue^ do
  begin
    Result := IntToStr(dwFileVersionMS shr 16);
    Result := Result + '.' + IntToStr(
      dwFileVersionMS and $FFFF);
    Result := Result + '.' + IntToStr(
      dwFileVersionLS shr 16);
    Result := Result + '.' + IntToStr(
      dwFileVersionLS and $FFFF);
  end;
  FreeMem(VerInfo, VerInfoSize);
end;
```

13. Debugger – Adicionando condição a um Breakpoint

Um dos principais recursos disponíveis no *debugger* do Delphi sem dúvida alguma é o *breakpoint*. Através dele, podemos fazer com que nossa aplicação pare em um determinado ponto do código para que possamos depurá-lo a fim de encontrar um erro ou uma falha na lógica.

Porém, existem situações onde o problema encontra-se dentro de um loop (comando *while..do* ou *for..do*) , e ao adicionarmos o *breakpoint* temos que depurar dentro do loop utilizando as teclas F7 ou F8 até o momento em que o erro ocorre.

O Delphi permite definir uma condição para a parada do *breakpoint* dentro do código onde, quando tal condição for verdadeira, o aplicativo pára na linha em questão. Veja o código de exemplo da **Listagem 7**.

Listagem 7. Exemplo para uso do breakpoint com condição

```
procedure TForm1.btnExecutarClick(
  Sender: TObject);
var
  x, y: Integer;
begin
  x := 1;
  y := 1;
  while x < 100 do
  begin
    y := y + x;
    inc(x);
  end;
  Edit1.Text := IntToStr(y);
end;
```

Vamos adicionar um *breakpoint* na linha *y := y + x;*. Digamos que um erro está ocorrendo quando o valor de *y* é igual a 16. Nesse caso, deveríamos ir executando a aplicação utilizando a tecla F8 (*Step Over*), correto? Bem, nesses casos podemos definir uma condição do tipo (*y = 16*), onde somente quando a mesma for verdadeira o *breakpoint* para na linha de código.

Para isso, selecione a opção *View>Debug Windows> Breakpoints* para abrir a janela *Breakpoint List*. Selecione o *breakpoint* em questão e acesse a opção *Properties* disponível no menu de contexto. Dentro da janela *Source Breakpoint Properties* adicione a condição "y = 16", dentro do campo *Condition* (**Figura 4**).

Ao rodar a aplicação o *breakpoint* só irá parar na linha de código quando a condição for verdadeira, evitando assim a depuração desgastante através da tecla F8 dentro do loop.

14. Trabalhando com packages em runtime

Caso sua aplicação esteja ficando com o tamanho do arquivo executável muito grande, você pode diminuí-lo utilizando a opção de *packages* em runtime. Quando criamos uma aplicação Delphi, por menor que ela seja, ao gerarmos o arquivo executável, todos os pacotes os quais contêm os componentes que utilizamos na aplicação são embutidos dentro do arquivo.

Podemos distribuir esses pacotes separadamente da aplicação, fazendo com que dessa maneira nosso arquivo executável fique menor. Entre na janela *Options* do projeto (*Project>Options*) e dentro da aba *Packages* marque a opção *Build with runtime packages*.

Rode a aplicação. O arquivo executável gerado será bem menor, entretanto, os pacotes dos componentes que foram utilizados na aplicação terão que ser distribuídos junto com o executável para que a aplicação funcione.

Para saber quais pacotes (arquivos *bpl*) devem ser levados junto com o executável da aplicação acesse a opção *Project>Information for NomeProjeto* (**Figura 5**).

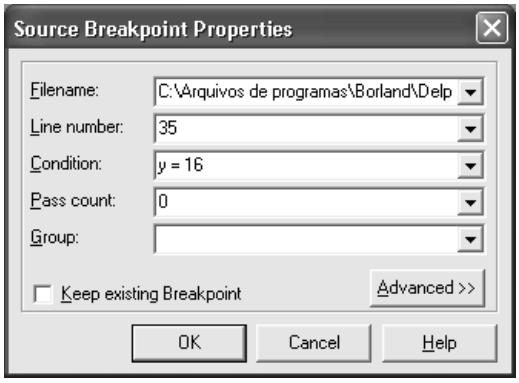


Figura 4. Definindo uma condição para a parada do breakpoint

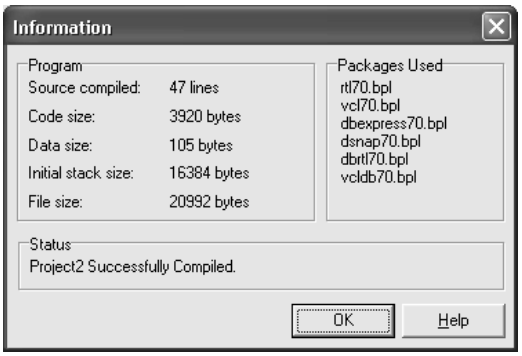


Figura 5. Verificando a lista dos packages utilizados pela aplicação

15. Enviando e-mail com anexo utilizando componentes Indy

Nessa última dica, veremos com enviar e-mails com arquivos em anexo utilizando os componentes *Indy*. Para enviar e-mails a partir de uma aplicação Delphi adicione um *IdSMTP* (*Indy Clients*) e um *IdMessage* (*Indy Misc*). Adicione ainda um botão e no seu evento *OnClick* digite o código da **Listagem 8**.

Listagem 8. Enviando e-mail com anexo

```
procedure TForm1.btnEnviarClick(Sender: TObject);
begin
  IdMessage1.From.Address :=
    'everson@rhealeza.com.br';
  IdMessage1.Recipients.EMailAddresses :=
    'joao.silva@gmail.com';
  IdMessage1.Subject := 'Assunto - Enviando e-mails '+
    'com anexo utilizando Indy';
  IdMessage1.Body.add('Corpo do email');
  IdMessage1.ContentType := 'Text';
  TIdAttachment.Create(IdMessage1.MessageParts,
    'c:\temp\Arquivo.txt');
  IdSMTP1.host := 'seuhost';

  // caso o servidor use autenticação:
  IdSMTP1.Password := 'senha';
  IdSMTP1.Username := 'usuario@dominio';
  IdSMTP1.AuthenticationType := atLogin;

  IdSMTP1.Connect;
  try
   IdSMTP1.send(IdMessage1);
  finally
   IdSMTP1.Disconnect;
  end;
end;
```

Conclusão

Vimos neste artigo várias dicas que podem ser utilizadas no dia a dia do desenvolvedor Delphi. Algumas dessas dicas podem ser consideradas básicas para alguns, porém, a muitos novos desenvolvedores começando a trabalhar com Delphi, onde tenho certeza que elas poderão ajudá-los durante o desenvolvimento e aprendizado. Um abraço e até a próxima. ■

Aplicações MultiBanco

Adapte sua aplicação para funcionar com diferentes BDs, usando técnicas de desenvolvimento em camadas e POO

CESAR BLUMM
(cesarblumm@yahoo.com.br)
é tecnólogo em Processamento de Dados pela Universidade de Caxias do Sul (UCS) e pós-graduado em Banco de Dados pela Universidade Luterana do Brasil (ULBRA), Certificação MCP, trabalha em desenvolvimento há 19 anos, atualmente é DBA na Metadados Assessoria e Sistemas trabalhando com Oracle, MS SQL Server e Firebird e desenvolve projetos na plataforma Delphi.

MIGUEL RODRIGUES FORNARI
(fornari@ieee.org) é bacharel e mestre pela Universidade do Rio Grande do Sul (UFRGS) e doutorando na área de Sistemas de Informação Geográfica pela mesma universidade. Professor da Universidade Luterana do Brasil (ULBRA) e Faculdades SENAC/RS. Também é consultor de diversas empresas na área de administração de SGBD.

FRANCISCO M. TRINDADE
(fmtrindade@m3tech.com.br)
é Engenheiro de Computação pela Universidade Federal do Rio Grande do Sul (UFRGS) e mestrando na área de Engenharia de Software pela mesma universidade. Atualmente trabalha com o desenvolvimento de soluções em Delphi e Firebird, pela empresa M3Tech Tecnologia para o Agronegócio.

Toda *software house*, ao desenvolver seus produtos, tem que viabilizar seus softwares para serem adquiridos pelo número máximo possível de clientes, independente do SGBD que o cliente utilizar. O objetivo deste artigo é mostrar como desenvolver software nessas condições, escrevendo o código-fonte uma única vez, e que possa funcionar para diferentes SGBDs, como Oracle, SQL Server, Firebird e PostgreSQL.

Para tanto, é necessário resolver problemas de compatibilidade entre os diferentes SGBDs, como tipos de dados e funções. Também deve-se criar uma arquitetura incluindo uma camada de persistência, capaz de gerenciar adequadamente o acesso a dados.

Criação de tabelas

Todos os fornecedores de BDs se propõem a armazenar os principais tipos representáveis de dados em tabelas, sendo que alguns têm mais opções do que os outros. Quem trabalha com mais de um fornecedor, provavelmente já investiu um tempo para pesquisar as ferramentas CASE ou para modelagem de diagramas E-R. A maioria delas possui o recurso de traduzir o modelo especificado para os diferentes fornecedores de SGBD, respeitando as suas diferenças.

Para criar uma ferramenta desse tipo para a sua aplicação, você terá que em um primeiro momento delimitar quais os bancos de dados que pretende trabalhar, e realizar um estudo dos tipos compatíveis entre eles. Na **Tabela 1** estão relacionados os tipos mais comuns em cada banco de dados e o equivalente nos outros SGBDs tratados neste artigo.

É importante considerar que além dos tipos de dados relacionados na **Tabela 1**, também existem tipos similares, que possuem funções específicas. Por exemplo: o tipo *nvarchar* no SQL Server também armazena valores alfa-

Tipo	Oracle	MS SQL Server	Firebird	PostgreSQL
Alfanumérico - tamanho variável	Varchar2	Varchar	Varchar	Varchar
Alfanumérico - tamanho fixo	Char	Char	Char	Char
Binário	Long Raw	Image	Blob	Bytea
Data/Hora	Date	DateTime	Date	Date
Inteiro	Int/Integer	Int	Integer	Int8
Numérico com Decimais	Number	Decimal	Decimal	Numeric
Texto	Long	Text	Varchar	Text

Tabela 1. Tipos de dados dos SGBDs

numéricos, porém usa a codificação *Unicode*, usada para representação de caracteres de línguas específicas como o japonês. Quanto à sintaxe da instrução *Create*, quando forem usadas as opções básicas da instrução, o comando é padrão para todos os bancos, conforme exemplo na **Listagem 1.**

Listagem 1. Instrução de criação de uma tabela (tipos do SQL Server)

```
Create Table Cliente
  (ID        Int           Not Null,
   Codigo    Int           Not Null,
   Nome      Varchar(40)   Not Null,
   Endereco Varchar(40), Constraint PK_Cliente Primary Key(ID));
```

A manutenção das tabelas utiliza praticamente uma sintaxe padrão. Uma exceção são as versões anteriores do Oracle, que não permitem retirar um atributo da mesma, sendo necessário salvá-la em outra temporária, recriá-la sem o campo e restaurar as informações para a tabela original.

Triggers/Stored Procedures

A vantagem de se utilizar *Triggers* e *Stored Procedures* é que você consegue colocar as regras do negócio da aplicação no banco de dados, agilizando alguns procedimentos e, principalmente, protegendo a integridade dos dados independentemente do aplicativo que está acessando as informações. Entretanto a portabilidade da aplicação é inversamente proporcional ao uso dos mesmos.

As instruções SQL são padronizadas pelo padrão ANSI e por isso possuem uma boa similaridade entre os diversos fornecedores de SGBD, porém a linguagem empregada no desenvolvimento de *Triggers* e *Stored Procedure* não é controlada por nenhum padrão e cada fornecedor possui uma linguagem que geralmente é incompatível com os outros fornecedores.

Resumindo você terá que dar manutenção em tantas fontes quantos forem os bancos de dados onde a sua aplicação irá executar. Na **Listagem 2** são apresentadas *Triggers* que fazem exatamente a mesma operação, porém observe a sintaxe utilizada por cada fornecedor (neste exemplo citei o Firebird e SQL Server, no endereço para download você encontra também o código para Oracle e PostgreSQL). Todas as *Triggers* controlam a inclusão de um item da nota fiscal, limitando a um máximo de 30 itens.

Listagem 2. Comandos para criação de Triggers em diferentes banco de dados

Trigger para o SQL Server

```
-- Exclui a trigger caso ela já exista
If Object_id ('TR_IN_ITEM','TR') Is Not Null
   Drop Trigger TR_IN_ITEM
Go
Create Trigger TR_IN_ITEM
   On  ITEM
   After Insert
as
Declare
    @LTOTALITENS INT,
    @LNOTA INT
Begin
  -- Seleciona o código da nota
  que está sendo incluída.
    Select @LNOTA = ID
      From INSERTED;
  -- Totaliza a quantidade de itens
  já cadastrados na nota.
    Select @LTOTALITENS = COUNT(*)
      From ITEM
     Where ID = @LNOTA;
  -- Verifica se existem mais do que 30 itens na nota.
    If @LTOTALITENS > 30
    Begin
       Raiserror('A nota %d já está cheia. '+
         'Inclusão cancelada.', 16, 1, @LNOTA);
       Rollback;
    End
End
Go
```

Trigger para o Firebird

```
Create Exception Nota_Cheia
  'A nota já está cheia. Inclusão cancelada.';

Create Trigger TR_IN_ITEM
   For  ITEM
   After Insert
as
Declare LTOTALITENS INTEGER;
Declare LNOTA INTEGER;
Begin
  /* Seleciona o código da nota
  que está sendo incluída. */
    LNOTA = New.ID;
  /* Totaliza a quantidade de itens
  já cadastrados na nota. */
    Select COUNT(*)
      From ITEM
     Where ID = :LNOTA
      Into :LTOTALITENS;
  /* Verifica se existem mais do que
  30 itens na nota. */
    If (LTOTALITENS > 30) Then
    Begin
       Exception Nota_Cheia;
    End
End
```

Comandos Insert/Update/Delete

Os comandos para inclusão, alteração e exclusão de dados no Delphi normalmente são geradas pelos próprios componentes utilizados para manipular as tabelas do banco de dados, como por exemplo o dbExpress. Porém em si-

tuações específicas, fornecer explicitamente os comandos *Insert, Update* e *Delete* pode ser bastante útil.

As sintaxes normalmente são bem similares. Uma das diferenças encontradas é que no SQL Server e no PostgreSQL não é permitido usar "apelidos" (aliases) no nome da tabela nas instruções *Update* e *Delete*. No código a seguir, vemos um exemplo da instrução no Oracle, com o uso do apelido *Cli* para a tabela, o que tornaria o código não-portável.

```
Update Cliente as Cli
   Set Nome = 'Teste'
 Where Exists
    (Select 1 from NotaFiscal as Nota
    Where Cli.ID = Nota.IDCliente)
```

Consultas SQL (Select)

As consultas SQL são essenciais aos nossos sistemas e, a princípio, todos SGBD atendem ao padrão *ANSI*. Porém, os fabricantes de SGBD, para ter um diferencial frente aos seus concorrentes, provêm seus produtos com extensões ao padrão. Essas extensões são muito úteis em algumas situações, porque podem reduzir a codificação necessária e facilitam a apresentação das informações nas aplicações. Entretanto, por elas não serem padronizadas, obrigam o desenvolvedor a escrever versões diferentes e não portáveis das consultas, sendo assim desaconselháveis. Ou seja, muitas vezes deixamos de usufruir de um poderoso recurso oferecido pelo BD, simplesmente para deixar nossa solução portável.

O caso mais comum é o das funções. Afora as básicas, que estão previstas no padrão, como *Sum, Count, Min, Max* e *Avg*, os SGBDs oferecem várias opções, como pode ser visto na **Tabela 2**, onde estão o objetivo e as versões para cada um dos SGBDs examinados neste artigo.

A solução básica é não incluir a função na consulta SQL, deixando para processar a informação com as funções existentes no Delphi. Essa solução é prática para funções na cláusula *Select*, mas inviável para a cláusula *where*.

Existem outras alternativas para realizar operações semelhantes às apresentadas, aqui apresentamos a forma mais usual. Por exemplo, o formato usado no Firebird para extrair dia, mês e ano de uma data com o *Extract*, também existe no Oracle.

Uma observação importante para quem pretende portar a aplicação para o PostgreSQL é quanto aos "apelidos" nos nomes das colunas do *Select*. Nele, a palavra-chave "as" é obrigatória, enquanto nos demais é opcional. No código a seguir, temos um exemplo que irá funcionar nos quatro SGBDs:

```
Select ID as Cli, Nome as RazaoSocial
From Cliente
```

Em *subselects* no Firebird o uso de apelidos é obrigatório quando é usada a mesma tabela na consulta principal e no *subselect*. No Oracle e no SQL Server não há essa regra. Na **Listagem 3**, a referência à tabela *Cliente* na linha 5 é interpretada no Firebird como a tabela declarada na linha 4 e não como a tabela da linha 2.

Listagem 3. Consulta com Subselect

```
1 Select ID, Nome
2   From Cliente
3  Where Exists
4     (Select 1 From Cliente Cli2
5       Where Cliente.ID = Cli2.ID
6         And Cli2.Endereco Is Null)
```

No Oracle e no SQL Server, o interpretador entende que *Cliente* diz respeito a tabela declarada na linha 2. Para resol-

Objetivo	Oracle	SQL Server	Firebird	PostgreSQL
Selecionar um pedaço de uma string	Substr(coluna, pos. Inicial, Tamanho)	Substring(coluna, pos.Inicial, Tamanho)	Substring(coluna From pos.Inicial For Tamanho)	Substring(coluna, pos.Inicial, Tamanho) Substr(coluna, pos.Inicial, Tamanho, Substring(coluna from pos.Inicial for Tamanho)
Separar o ano de uma data	To_Date(Coluna, 'YYYY')	Year(Coluna)	Extract(Year From Coluna)	Extract(Year From Coluna)
Separar o mês de uma data	To_Date(Coluna, 'MM')	Month(Coluna)	Extract(Month From Coluna	Extract(Month From Coluna
Separar o dia de uma data	To_Date(Coluna, 'DD')	Day(Coluna)	Extract(Day From Coluna)	Extract(Day From Coluna)
Resto da divisão	MOD(Coluna1, Coluna2)	Coluna1 % Coluna2 (Operador)		Coluna1 % Coluna2 (Operador)
Data corrente	Current_Date, Current_ TimeStamp	Current_TimeStamp	Current_ TimeStamp	Current_TimeStamp
Concatenação	String || String	String + String	String + String	String || String
Arredondamento	Round(Valor, Nro. Decimais)	Round(Valor, Nro. Decimais)	Não existe	Round(Valor, Nro.Decimais)

Tabela 2. Uso de funções nos SGBD

ver esse problema de portabilidade você precisa informar um apelido na tabela da linha 2 conforme a **Listagem 4.**

Listagem 4. Consulta com subselect portável para os quatro SGBD

```
1 Select ID, Nome
2   From Cliente As Cli1
3  Where Exists
4     (Select 1 From Cliente Cli2
5       Where Cli1.ID = Cli2.ID
6         And Cli2.Endereco Is Null)
```

Caso você pretenda portar a aplicação para o Oracle versão 8, será necessário alterar a sintaxe das junções, porque não existem as opções *Inner Join* e *Outer Join*. No caso do *Inner Join* é mais simples, pois basta colocar a comparação do *join* na cláusula *where* e o nome da tabela no *From*, conforme exemplo da **Listagem 5.**

Listagem 5. Troca de sintaxe da consulta de Inner Join para Where

```
Select NotaFiscal.ID, Cliente.Nome
From NotaFiscal
Inner Join Cliente on
   NotaFiscal.IDCliente = Cliente.ID

Alterar para

Select NotaFiscal.ID, Cliente.Nome
From NotaFiscal, Cliente
Where NotaFiscal.IDCliente = Cliente.ID
```

No caso do *Outer Join,* a mudança é simples, porém a sintaxe não ajuda muito e o entendimento da cláusula não é tão fácil. Observe na **Listagem 6,** que além das alterações do *Inner Join* também foi acrescido o operador (+) à direita do campo *Departamento* da tabela *Departamentos* no *where*. Esse operador indica o *Outer Join.* O sinal (+) deve ficar ao lado da coluna onde a informação poderá faltar na tabela. Está sintaxe funciona somente para o Oracle.

Listagem 6. Troca de sintaxe da consulta de Outer Join

```
Select NotaFiscal.ID, Cliente.Nome
From NotaFiscal
Left Outer Join Cliente on NotaFiscal.IDCliente = Cliente.ID

Alterar para

Select NotaFiscal.ID, Cliente.Nome
From NotaFiscal, Cliente
Where NotaFiscal.IDCliente = Cliente.ID (+)
```

Tabela de Caracteres

Na criação do banco de dados existe a possibilidade de informar qual a tabela de caracteres será utilizada. Essa tabela de caracteres identifica quais os caracteres são válidos para serem armazenados, e está associada a língua utilizada por quem irá gerar informações para o sistema.

Uma vez definida qual será a tabela utilizada, existem várias restrições feitas pelos SGBD para trocar essa opção. Na melhor hipótese será possível apenas trocar para outra tabela similar a anterior, com risco do SGBD trocar acentos e coisas do gênero.

Se o banco de dados for instalado com as opções padrões dos SGBDs, existem situações que geram resultados diferentes entre os fornecedores que merecem a nossa atenção, por exemplo, quando classificamos uma consulta SQL com a cláusula *Order By* com um campo que possua conteúdo nulo em algumas linhas da tabela. No Oracle e no Firebird a ordem dessas linhas será depois de todas as letras e números, já no SQL Server vem antes.

Também conforme a configuração da tabela de caracteres, as letras maiúsculas e minúsculas terão ou não distinção. Por exemplo, se você criar duas tabelas onde o campo chave é alfanumérico e a tabela detalhe possuir uma chave estrangeira para a tabela mestre, mesmo assim será possível ocorrer o seguinte tipo de situação:

```
Chave da Tabela Mestre: 'CD01'
Chave da Tabela Detalhe: 'cd01'
```

Observe que na tabela mestre o código está em maiúsculo e no detalhe o código está em minúsculo, o banco gravará dessa forma nas respectivas tabelas sem problema nenhum. Quando for montada uma instrução SQL selecionando essa coluna tanto faz se o argumento for "CD01", "cd01", "Cd01" ou "cD01" a linha será encontrada. Porém se na sua aplicação você carregar um cursor com várias linhas da tabela e for interpretar a seleção dentro do código Delphi, para o Delphi maiúsculo sempre será diferente de minúsculo, claro.

Outra situação importante nesse caso é quando for necessário converter um banco ou uma tabela com uma configuração onde exista distinção de maiúsculo e minúsculo para outro onde não exista essa distinção, e nela já existirem duas linhas uma em um formato e outra em outro formato, por exemplo: "CD01" e "cd01", neste caso quando for incluir a segunda linha nesse banco de dados, ocorrerá um erro de chave primária duplicada.

Essa configuração é padrão no SQL Server, nos outros bancos todos são instalados fazendo distinção de maiúsculas e minúsculas.

Camada de persistência

Uma possível solução para a utilização de SGBDs de diferentes fornecedores é a criação de uma camada de persistência independente, que seja responsável pela criação dos comandos SQL específicos de cada fornecedor, deixando a camada de negócio da aplicação isolada da solução de banco de dados. Será demonstrado neste artigo um exemplo de uma camada de persistência simples, que pode ser utilizada com esse objetivo.

Para a implementação da camada de persistência, será utilizado o padrão de projeto *Abstract Factory*, que estabelece uma interface abstrata para uma classe, que será instanciada através de classes descendentes. O diagrama de classes do padrão *Abstract Factory* é mostrado na **Figura 1**.

No modelo que implementaremos será criada uma classe abstrata de persistência, que terá em sua interface as funções de *Insert*, *Delete,Update* e *Select*, de forma que a aplicação possa, através dessa interface, realizar as operações necessárias com o banco de dados.

As funções declaradas na interface dessa classe de persistência genérica serão implementadas em classes de persistência descendentes, uma para cada tipo de banco de dados que necessitar ser utilizado. O diagrama de classes da camada de persistência proposta é mostrado na **Figura 2.**

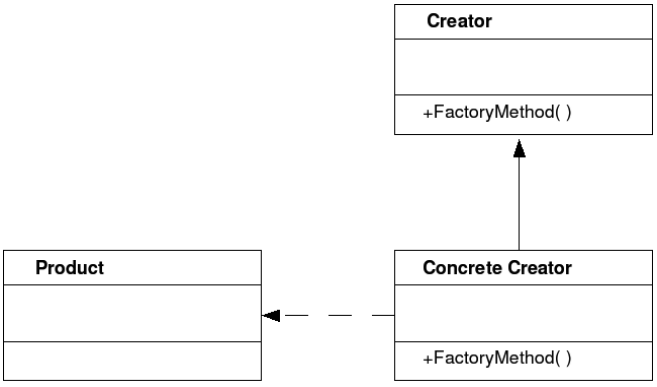


Figura 1. Diagrama de classes Abstract Factory

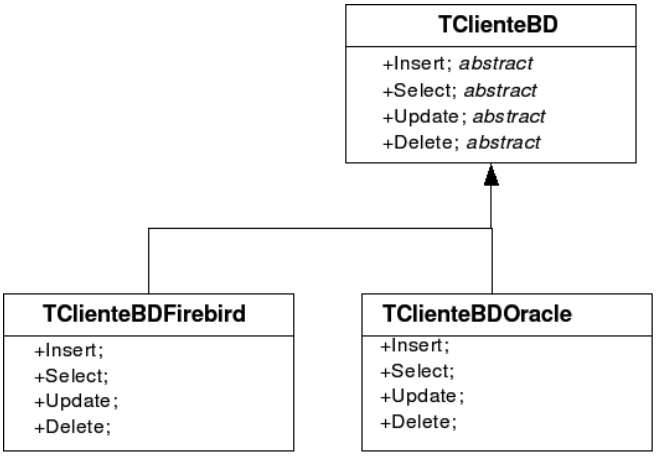


Figura 2. Diagrama de classes da camada de persistência independente de banco de dados

Exemplo

De forma a exemplificar a criação e utilização da camada de persistência proposta, foi criado um pequeno aplicativo, que possui os conceitos de *Item, Cliente* e *Nota Fiscal*, sendo que a *Nota Fiscal* possui ligação com um *Cliente* e diversas instâncias de *Item.* A **Figura 3** mostra o modelo conceitual do exemplo implementado.

Para a implementação do banco de dados do aplicativo, serão necessárias três tabelas, cada uma representando um dos conceitos propostos anteriormente. A estrutura de cada tabela do banco de dados é mostrada na **Tabela 3**.

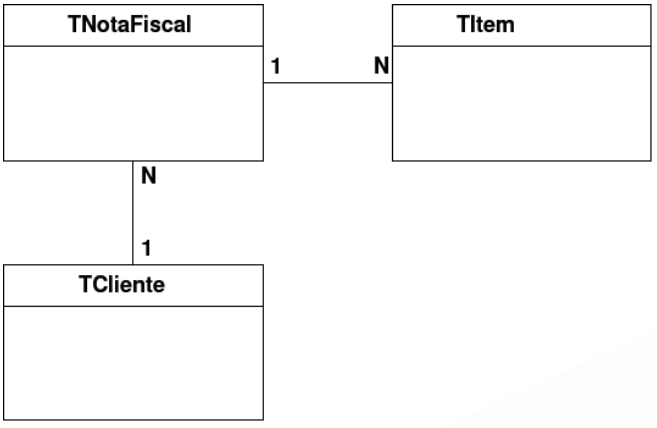


Figura 3. Modelo Conceitual do aplicativo

Tabela	Campos
Cliente	Id, Codigo, Nome e Endereco
Item	Id, Codigo, Preco, Quantidade e IdNotaFiscal
Nota Fiscal	Id, Codigo e IdCliente

Tabela 3. Estrutura das tabelas do aplicativo-exemplo

Seguindo o modelo proposto, para cada tabela especificada, será criada uma classe abstrata de persistência, responsável por determinar a interface a ser utilizada pelas classes descendentes.

Dessa forma, foram criadas as classes *TClienteBD, TItemBD e TNotaFiscalBD*, que são responsáveis pela persistência das classes de negócio *TCliente, TItem* e *TNotaFiscal*, respectivamente. O código das classes *TCliente* e *TClienteBD* é mostrado na **Listagem 7.**

Listagem 7. Código de implementação das classes TCliente e TClienteBD

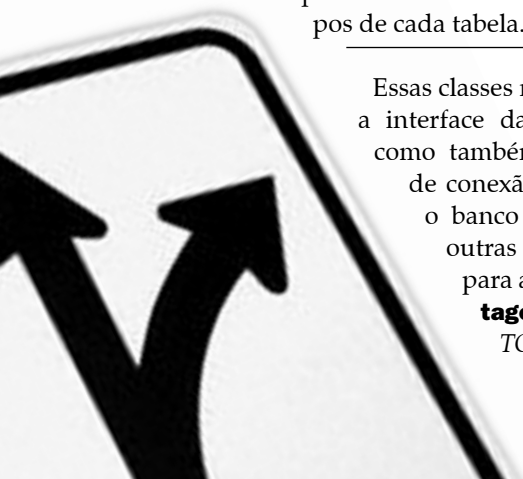
```
unit uClasses;

interface

uses
  Classes, ContNrs;

type
  TCliente = class
  private
    FId: Integer;
    FCodigo: Integer;
    FEndereco: string;
    FNome: string;

  public
    property Id: Integer read FId write FId;
    property Codigo: Integer read FCodigo
      write FCodigo;
    property Nome: string read FNome write FNome;
    property Endereco: string read FEndereco
      write FEndereco;
  end;


  TClienteBD = class
  public
    function Insert(ACodigo: Integer; ANome,
      AEndereco: string): TCliente; virtual; abstract;
    procedure Delete(var ACliente: TCliente);
      virtual; abstract;
    procedure Update(ACliente: TCliente);
      virtual; abstract;
    function  Select(CondicoesSQL: string):
      TObjectList; virtual; abstract;
  end;
```

O exemplo pode ser criado em qualquer versão do Delphi. A classe *TCliente* possui somente os campos *Codigo, Endereco* e *Nome*, que definem o cliente, e o campo Id, que é utilizado como chave primária no banco de dados.

A classe *TClienteBD* é uma classe abstrata, portanto contém somente a declaração dos métodos necessários para acesso ao banco de dados, que são declarados com as diretivas *virtual* e *abstract,* deixando a implementação para ser realizada pelas classes descendentes.

O primeiro tipo de banco de dados escolhido para o aplicativo foi o Firebird. Assim, foi criada uma implementação concreta das classes de persistência do sistema em implementação, chamadas de *TClienteBDFirebird, TItemBDFirebird* e *TNotaFiscalBDFirebird*.

Nota: O código completo das classes *TItem* e *TNotaFiscal*, assim como as implementações das classes descendentes, está disponível para download. Mas sua concepção é bastante semelhante ao já apresentado neste artigo, diferenciando apenas pela quantidade e nomenclatura dos campos de cada tabela.



Essas classes não só implementam a interface da classe ascendente, como também possuem funções de conexão e desconexão com o banco de dados, além de outras rotinas necessárias para a sua execução. **A Listagem 8** mostra a classe *TClienteBDFirebird*.

Listagem 8. Classe TClienteBDFirebird

```
uses SqlExpr, Provider;
...

TClienteBDFirebird = class(TClienteBD)
  private
    BdLib: string;
    Bd: string;
    Password: string;
    Username: string;
    AppPath: string;

    { Variáveis de conexão }
    dbExpConnection: TSQLConnection;
    dbExpQuery: TSQLQuery;
    DbExpDataSetProvider: TDataSetProvider;
    GeneratorQuery: TSQLQuery;

    procedure Conectar;
    procedure Desconectar;
    procedure ExecutarQuerySQL(stringSQL: string);
    function  SelectQuerySQL(
      stringSQL: string): Integer;
    function  GetValorGenerator() : Integer;
  public
    constructor Create;
    destructor Destroy; override;

    function Insert(ACodigo: Integer; ANome,
      AEndereco: string): TCliente; override;
    procedure Delete(var ACliente: TCliente); override;
    procedure Update(ACliente: TCliente); override;
    function Select(CondicoesSQL: string):
      TObjectList; override;
  end;
```

A **Listagem 9** mostra a implementação dos métodos *Conectar* e *Desconectar* da classe *TClienteBDFirebird*, que são invocados no *Create* e *Destroy* da classe, respectivamente.

Na **Listagem 10** temos a implementação dos outros métodos da classe *TClienteBDFirebird*.

A classe deve ser definida na unit *uCliente*, criada anteriormente. A classe criada deve implementar os métodos abstratos declarados na classe *TClienteBD,* usando-se a diretiva *override* nas funções abstratas que serão sobrescritas.

Essa função recebe os parâmetros do objeto a ser criado, monta a instrução SQL para execução no banco e realiza a inserção. Após a operação, um objeto do tipo *TCliente* é criado com os parâmetros utilizados. O método *GetValorGenerator* é responsável por verificar o valor da chave primária, executando um comando *select* no *Generator* criado no banco (nesse caso, GEN_CLIENTE_ID).

Nota: A implementação das classes dos outros tipos de banco de dados, assemelha-se em muito com a classes criadas anteriormente; a modificação refere-se apenas as características de cada banco de dados (como por exemplo, o método *Conectar*).

Apresentação dos dados do banco

Com as camadas de persistência e negócio prontas, foi criado um formulário para apresentação do aplicativo desenvolvido. Esse formulário possui somente ligação com as classes de persistência abstratas, isolando-se da implementação específica da persistência do programa.

No momento de ativação do formulário, ele então instancia a camada de persistência específica que será utili-

zada, através de um tipo definido na aplicação, sendo esse a única parte da camada superior do programa que teria que ser modificada no caso de mudança no tipo de banco de dados utilizado.

Listagem 9. Métodos para conexão e desconexão com o banco de dados

```
procedure TClienteBDSQLServer.Conectar;
begin
  try
    BdLib := '<caminho>\fbclient.dll';
    Bd := '<caminho>\ARTIGO.FDB';
    Password := 'masterkey';
    Username := 'SYSDBA';

    dbExpConnection := TSQLConnection.Create(
      dbExpConnection);
    dbExpQuery := TSQLQuery.Create(dbExpQuery);
    DbExpDataSetProvider := TDataSetProvider.Create(
      DbExpDataSetProvider);

    with dbExpConnection do
    begin
      DriverName := 'INTERBASE';
      VendorLib := BdLib;
      ConnectionName := 'IBConnection';
      LibraryName := 'dbexpint.dll';
      GetDriverFunc := 'getSQLDriverINTERBASE';

      Params.Add('SQLDialect=3');
      Params.Add('Database='+ Bd);
      Params.Add('user_name=' + Username);
      Params.Add('password='+ Password);
      LoginPrompt := False;
      Connected := True;
    end;

    dbExpQuery.SQLConnection := dbExpConnection;

  { Query para recuperar o Id do objeto criado }
  GeneratorQuery := TSQLQuery.Create(GeneratorQuery);
  GeneratorQuery.SQLConnection := dbExpConnection;

  with DbExpDataSetProvider do
  begin
    DataSet := dbExpQuery;
    Name := 'DbExpProvider';
  end;

  except
  on E: Exception do
    raise EAbort.Create(
      'TClienteBDFFirebird.Conectar: Erro conectando');
  end;
end;

procedure TClienteBDFirebird.Desconectar;
begin
try
  if dbExpConnection <> nil then
  begin
    dbExpConnection.Connected := False;
    FreeAndNil(dbExpQuery);
    FreeAndNil(dbExpConnection);
    FreeAndNil(DbExpDataSetProvider);
  end
  else
   raise EAbort.Create(
     'TClienteBDFFirebird.Desconectar: '+
     'Já desconectado');
except
  on EAbort do
  begin
    raise;
  end;
end;
end;
```

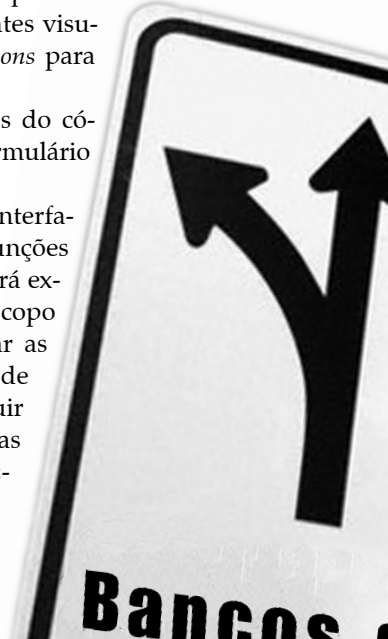
Listagem 10. Outras implementações da classe TClienteBDFirebird

```
function TClienteBDFirebird.Insert(ACodigo: Integer;
  ANome, AEndereco: string): TCliente;
var
  SQLString: string;
  VCliente: TCliente;
begin
  SQLString := 'INSERT INTO CLIENTE VALUES (-1,';
  SQLString := SQLString + IntToStr(ACodigo) + ',';
  SQLString := SQLString + QuotedStr(ANome) + ',';
  SQLString := SQLString + QuotedStr(AEndereco) + ');';
  ExecutarQuerySQL(SQLString);

  VCliente := TCliente.Create;
  VCliente.Id := Self.GetValorGenerator;
  VCliente.Codigo := ACodigo;
  VCliente.Nome := ANome;
  VCliente.Endereco := AEndereco;
  Result := VCliente;
end;

procedure TClienteBDFirebird.Delete(
  var ACliente: TCliente);
var
  SQLString: string;
begin
  SQLString := 'DELETE FROM CLIENTE WHERE ID = ' +
    IntToStr(ACliente.Id) + ';';
  ExecutarQuerySQL(SQLString);
  FreeAndNil(ACliente);
end;

procedure TClienteBDFirebird.Update(
  ACliente: TCliente);
var
  SQLString: string;
begin
  SQLString := 'UPDATE CLIENTE SET ' +
  'CODIGO = ' + IntToStr(ACliente.Codigo) + ',' +
  'NOME = '   + QuotedStr(ACliente.Nome) + ',' +
  'ENDERECO = '+ QuotedStr(ACliente.Endereco) +
  ' WHERE ID = ' + IntToStr(ACliente.FId);
  ExecutarQuerySQL(SQLString);
end;

function TClienteBDFirebird.Select(
  CondicoesSQL: string): TObjectList;
var
  SQLString: string;
  VClientes: TObjectList;
  VClienteTemp: TCliente;
  VCount: Integer;
begin
  SQLString := 'SELECT * FROM CLIENTE WHERE ' +
    CondicoesSQL + ';';
  SelectQuerySQL(SQLString);
  VClientes := TObjectList.Create(True);
  for VCount := 0 to
    Self.dbExpQuery.RecordCount -1 do
  begin
    VClienteTemp := TCliente.Create;
    VClienteTemp.FId := Self.dbExpQuery.FieldByName(
      'ID').Value;
    VClienteTemp.FCodigo :=
      Self.dbExpQuery.FieldByName('CODIGO').Value;
    VClienteTemp.FNome := Self.dbExpQuery.FieldByName(
      'NOME').Value;
    VClienteTemp.FEndereco :=
      Self.dbExpQuery.FieldByName('ENDERECO').Value;
    VClientes.Add(VClienteTemp);
    dbExpQuery.Next;
  end;
  Result := VClientes;
end;
```

Para isso, devemos criar um formulário que conterá a interface de nossa aplicação exemplo. Adicione também alguns componentes visuais, como *ListBox*, *TextBox* e *Buttons* para iniciar o exemplo (**Figura 4**).

A **Listagem 11** mostra trechos do código de implementação do formulário principal da aplicação.

A criação dos componentes de interface, assim como declaração das funções de tratamento de eventos não será explicada aqui, por estar fora do escopo deste artigo. De forma a acessar as informações contidas no banco de dados, o formulário deverá possuir as classes de persistência criadas anteriormente, *TClienteBD, TItemBD e TNotaFiscalBD.*



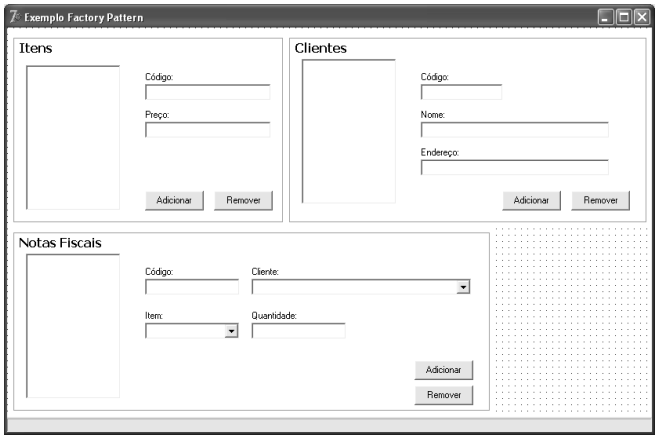


Figura 4. Sugestão de layout da aplicação

Listagem 11. Implementação do formulário principal da aplicação

```
uses uClasses; { unit das classes criadas }

type
  { Tipo enumerado para definição da
    camada de persistencia }
  TTipoBancoDeDados = (sgbdFirebird, sgbdSQLServer);

  TMainForm = class(TForm)
...
  { Campos independentes do tipo de SGBD }
  FClienteBD: TClienteBD;
...

procedure TMainForm.InicializarCamadaPersistencia(
  ATipo: TTipoBancoDeDados);
begin
  { Instância da camada de persistência de
    acordo com parâmetro }
  case ATipo of
    sgbdFirebird: FClienteBD :=
      TClienteBDFirebird.Create;
  end;
end;

procedure TMainForm.FormActivate(Sender: TObject);
begin
  { Classes específicas de persistência }
  Self.InicializarCamadaPersistencia(sgbdFirebird);
  { Inicializar os itens da NotaFiscal ou qualquer
   outra necessidade }
end;

procedure TMainForm.FormClose(
  Sender: TObject; var Action: TCloseAction);
begin
  Self.FClienteBD.Free;
end;
```

Deve-se observar que as classes declaradas no formulário são abstratas, sendo que na criação do objeto deveremos especificar o tipo de banco de dados que utilizaremos. Isso é feito no nosso código através da função *InicializarCamadaPersistencia*, que recebe como parâmetro o tipo de banco de dados desejado.

Trocando de SGBD

Para exemplificar a independência da camada de persistência proposta, vamos simular a troca do sistema de gerenciamento de banco de dados, do Firebird, que apresentamos anteriormente, para SQL Server.

No desenvolvimento da camada de persistência para o novo SGBD, iremos implementar novas classes específicas de persistência, chamadas de *TClienteBDSQLServer, TItemBDSQLServer* e *TNotaFiscalBDSQLServer*.

Essas classes suportarão as mesmas interfaces propostas pelas classes genéricas de persistência, adequando os parâmetros de conexão, assim como diferenças no SQL existentes no novo tipo de banco de dados. Como exemplo, a **Listagem 12** mostra a implementação do *Conectar* e *Insert* da classe *TClienteBDSQLServer.*

Listagem 12. Implementação da função TClienteBDSQLServer.Insert

```
procedure TClienteBDSQLServer.Conectar;
begin
  try
    Password := '$root';
    Username := 'root';

    dbExpConnection := TSQLConnection.Create(
      dbExpConnection);
    dbExpQuery := TSQLQuery.Create(dbExpQuery);
    DbExpDataSetProvider := TDataSetProvider.Create(
      DbExpDataSetProvider);

    with dbExpConnection do
    begin
      DriverName := 'MSSQL';
      VendorLib := 'oledb.dll';
      ConnectionName := 'MSSQLConnection';
      LibraryName := 'dbexpmss.dll';
      GetDriverFunc := 'getSQLDriverMSSQL';

      Params.Add('Database=tempdb');
      Params.Add('HostName=HOST\SQLEXPRESS');
      Params.Add('user_name=' + Username);
      Params.Add('password='+ Password);
      Params.Add('OS Authentication=True');
      LoginPrompt := False;
      Connected := True;
    end;

  dbExpQuery.SQLConnection := dbExpConnection;

  { Query para recuperar o Id do objeto criado }
  GeneratorQuery := TSQLQuery.Create(GeneratorQuery);
  GeneratorQuery.SQLConnection := dbExpConnection;

  with DbExpDataSetProvider do
  begin
    DataSet := dbExpQuery;
    Name := 'DbExpProvider';
  end;

  except
  on E : Exception do
    raise EAbort.Create(
      'TClienteBDSQLServer.Conectar: Erro conectando');
  end;
end;

function TClienteBDSQLServer.Insert(
  ACodigo: Integer; ANome,
  AEndereco: string): TCliente;
var
  SQLString: string;
  VCliente: TCliente;
begin
  SQLString := 'INSERT INTO [tempdb].[dbo].[CLIENTE] VALUES (';
  SQLString := SQLString + IntToStr(ACodigo) + ',';
  SQLString := SQLString + QuotedStr(ANome) + ',';
  SQLString := SQLString + QuotedStr(AEndereco) + ')';

  ExecutarQuerySQL(SQLString);

  VCliente := TCliente.Create;
  VCliente.Id := Self.GetValorId;
  VCliente.Codigo := ACodigo;
  VCliente.Nome := ANome;
  VCliente.Endereco := AEndereco;

  Result := VCliente;

end;
```

Pode-se observar que o comando SQL utilizado para a execução da inserção é diferente do utilizado anteriormente, pois o valor para ID, que anteriormente era -1 na implementação para o Firebird, aqui não é colocado. Como também o método *Conectar* que possui parâmetros diferentes dos utilizados para conectar ao Firebird.

> **Nota:** Uma outra característica da camada de persistência está na estrutura da tecnologia dbExpress, onde facilmente, somente alterando parâmetros, podemos conectar a outra base de dados.

Com a camada de persistência para o banco de dados SQL Server implementada, basta modificar o tipo de objeto de persistência que é instanciado no início do aplicativo, de forma rápida e fácil, conforme mostrado pela **Listagem 13.**

Listagem 13. Função de inicialização do aplicativo instanciando a camada de persistência para o BD SQLServer

```
procedure TMainForm.FormActivate(Sender: TObject);
begin
  { Self.InicializarCamadaPersistencia(sgbdFirebird); }
  Self.InicializarCamadaPersistencia(sgbdSQLServer);
  Self.InicializarGrpClientes;
end;

procedure TMainForm.InicializarCamadaPersistencia(
  ATipo: TTipoBancoDeDados);
begin
  { Instância camada de persistência de
    acordo com parâmetro }
  case ATipo of
    sgbdFirebird: FClienteBD := TClienteBDFirebird.Create;
    sgbdSQLServer: FClienteBD := TClienteBDSQLServer.Create;
  end;
end;
```

A variável *Atipo* pode ser um parâmetro de configuração do sistema, que indica o SGBD que está sendo utilizado. Após essa pequena modificação, o aplicativo estará pronto para ser executado com o banco de dados SQL Server. Veja na **Figura 5** o exemplo em execução.

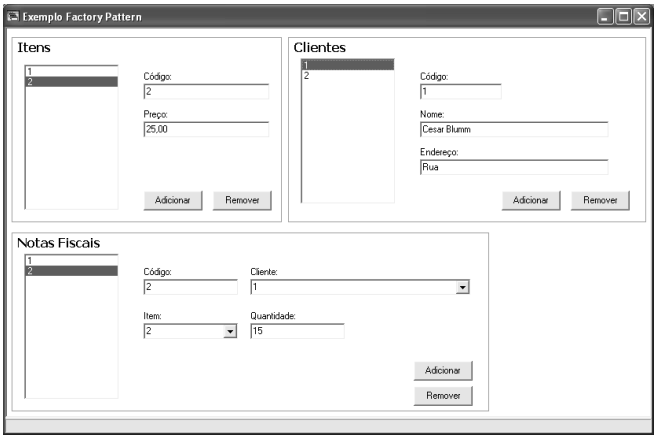


Figura 5. Projeto em execução, utilizando o Firebird

Conclusão

Apesar dos padrões existentes que regem a sintaxe do SQL, é um tanto questionável se esse é um padrão de fato, pois existem muitas diferenças entre os fornecedores de SGBD. Quando alguma empresa pretender portar a sua aplicação para mais do que um SGBD, provavelmente começarão os problemas com o sistema.

É muito importante que em tempo de projeto estejam previstos quais os bancos de dados alvos que serão utilizados pela aplicação, para que essas diferenças sejam tratadas adequadamente. Em tempo de desenvolvimento, sempre que possível, é interessante ir testando a aplicação em todos os bancos de dados onde ela executará, assim outros detalhes que existam entre os SGBD, já vão sendo tratados antes do término do projeto.

Além disso, foi apresentada a implementação de uma camada de persistência independente do sistema de banco de dados, utilizando-se do padrão de projeto *Abstract Factory.* Essa abordagem pode ser muito útil na implementação de aplicações para diferentes tipos de SGBD, possuindo a vantagem de manter a camada de negócios da aplicação isolada da implementação das rotinas de persistência, permitindo assim o reuso de código de aplicação.■

Links

Site do Firebird

www.ibphoenix.org

Site do Microsoft SQL Server

www.microsoft.com/sql

Site da Oracle

otn.oracle.com

Site do PostgreSQL

www.postgresql.org

Edições Anteriores
Complete a sua Coleção.

EDIÇÃO 51

- Novidades da Delphi Language: Parte IV
- Database Explorer
- POO na Prática
- Criando controles ASP.NET

EDIÇÃO 52

- Novidades da Delphi Language: Parte IV
- Database Explorer
- POO na Prática
- Criando controles ASP.NET

EDIÇÃO 53

- Novidades da Delphi Language: Parte IV
- Database Explorer
- POO na Prática
- Criando controles ASP.NET

EDIÇÃO 54

- Carga dinâmica de pacotes
- Plataforma .NET
- Segredos da VCL
- Threads e Concexões

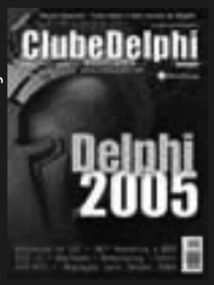
EDIÇÃO 55

- Web Services
- Guitar Explorer
- Rave Language
- Crystal Reports

EDIÇÃO 56

- Novidades da Delphi Language: Parte IV
- Database Explorer
- POO na Prática
- Criando controles ASP.NET

EDIÇÃO 57

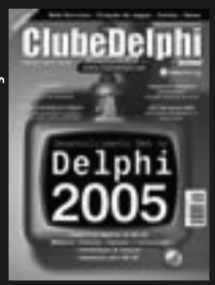


- Novidades da Delphi Language: Parte IV
- Database Explorer
- POO na Prática
- Criando controles ASP.NET

EDIÇÃO 58

- Novidades da Delphi Language: Parte IV
- Database Explorer
- POO na Prática
- Criando controles ASP.NET

EDIÇÃO 59



- Novidades da Delphi Language: Parte IV
- Database Explorer
- POO na Prática
- Criando controles ASP.NET

EDIÇÃO 60

- IBExpert Total
- Migrando para Delphi for .NET Parte I
- StarTeam passo a passo
- TestComplete, Testes de aplicações

EDIÇÃO 61

- Programação Orientada por Objetos
- Acesso ao InterBase e Firebird
- Crystal Reports, Relatórios mestre
- Parâmetros em relatórios Rave

EDIÇÃO 62

- TreeView com Dados
- Criação de Jogos com o Delphi Parte V
- Curso de ASP.NET com Delphi Parte VII
- CMMI, Quem precisa dele?

EDIÇÃO 63

- Sistema SysCash
- Longhorn, O Futuro do Windows
- Usando o Repeater e DataList
- O que é CodeDOM?

EDIÇÃO 64

- DevExpress QuantumGrid
- Structured Query Language
- Guia de referência Parte II
- Comandos DML; Crystal Reports

EDIÇÃO 66

- Novidades da Delphi Language: Parte IV
- Database Explorer
- POO na Prática
- Criando controles ASP.NET

EDIÇÃO 67

- Novidades da Delphi Language Parte III
- Namespaces
- Comandos DCL e views
- DataSnap e DCOM + Parte III

EDIÇÃO 68

- Novidades da Delphi Language: Parte IV
- Database Explorer
- POO na Prática
- Criando controles ASP.NET

EDIÇÃO 69

- Delphi 2006
- VCL - Novos Componentes
- Desenvolvimento com C++
- Padrões de Projeto e POO

ClubeDelphi

DevMedia GROUP

Acesse agora a nossa página de compra ou se preferir, entre em contato com a nossa Central de Atendimento através do telefone: (21) 2283-9012.

www.devmedia.com.br/anteriores

FABRICIO DESBESSEL

AJAX

Usando com o Delphi a biblioteca MagicAjax

FABRÍCIO DESBESSEL (fabricio.desbessel@terra.com.br) é professor de Linguagem de Programação do Curso Técnico em Informática do Colégio Frederico Jorge Logemann de Horizontina/RS e da FAHOR Faculdade Horizontina. Delphiano de coração está sempre disposto a provar que com o Delphi sempre teremos a melhor solução. Site www.fabricio.pro.br.

Existem inúmeras formas de utilizar a tecnologia AJAX em aplicações .NET, mas nenhuma apresenta tanta facilidade como a utilização do framework *MagicAjax.Net*. Esse framework é gratuito, está na versão beta e pode ser baixado no endereço oficial do projeto: *www.magicajax.net*.

Com a sua utilização, não é necessário ter o conhecimento de Java Script para criar as funções que enviam e buscam (*callback*) as informações sem dar um *refresh* total na página, diminuindo o tráfego de informações entre o servidor e os clientes, consumindo menos tempo.

Além disso, ele automaticamente apresenta a mensagem de *Loading* no canto superior direito, da mesma forma que acontece no *Gmail*. Neste artigo não vou entrar na parte teórica e introdutória sobre o AJAX, pois isso já foi tratado em outro artigo da revista Clube Delphi na edição 70. Para uma rápida introdução sobre a tecnologia, veja o box "O que é AJAX?".

O que é AJAX?

Quando navegamos em páginas Web, sempre que é necessário efetuar uma comunicação com o servidor (clique de um botão, por exemplo), há um intenso tráfego de dados e toda a página é sempre recarregada (refresh total). Com tecnologias como o AJAX (Asynchronous JavaScript and XML), podemos limitar os postbacks ao servidor, evitando refreshs totais da página. Podemos chamar métodos de forma assíncrona e então ajustar pequenas porções da tela, diminuindo o tráfego de dados na rede. Com isso, aplicações Web se tornam semelhantes a aplicações Desktop. Trabalhar com AJAX "puro" requer vasto conhecimento de Java Script e exige codificação exaustiva. Dessa forma, várias tecnologias e ferramentas definem frameworks para tornar o trabalho sobre o AJAX mais produtivo, em várias linguagens e plataformas diferentes (Java, .NET etc.). Um desses frameworks é o MagicAjax.

Criando uma consulta a dados com Firebird

No Delphi 2005 ou 2006, crie um novo projeto do tipo *ASP.NET Web Application - Delphi for .NET*. Preencha o *Name* com "Magico" e clique em OK. Vamos criar uma consulta de dados nas tabelas *Department* e *Employee* do banco *Employee.fdb*, através do *Firebird Data Provider* (veja box). Adicione um *FbConnection*, clique na propriedade *ConnectionString* e configure suas propriedades conforme a **Figura 1**.

Teste a conexão clicando no botão *Test* e depois clique no *Accept*. Agora adicione um *FbCommand*, configurando sua propriedade *Connection* para o *fbConnection1* e a propriedade *CommandText* conforme a **Listagem 1**.

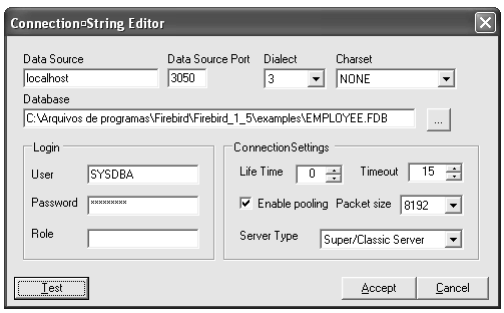


Figura 1. Configurações da conexão ao Firebird

Firebird Data Provider

Para acessar o Firebird a partir de aplicações ASP.NET, a melhor opção é usar o provider do banco para o ADO.NET, chamado Firebird Data Provider. Para instalá-lo, acesse o endereço *www.firebirdsql.com*, clique no link *Download* a seguir em *Firebird .NET Data Provider*. Baixe e instale a última versão do *Data Provider for .NET Framework 1.1*, bastando seguir os passos no assistente que será iniciado. Neste artigo usaremos versão 1.7 do *Provider*, versão mais recente disponível até o fechamento desta edição. Após a instalação, no Delphi 2005/2006 clique no menu *Component|Installed .NET Components*. Digite "Firebird Data Provider" na opção *Category*, clique no botão *Select an Assembly* e escolha o arquivo *FirebirdSql.Data.Firebird.dll*, localizado no diretório de instalação do *Provider*, por padrão em *C:\Arquivos de programas\FirebirdNETProvider(versão)*. Clique em *Ok* e observe que os novos componentes para acesso ao Firebird estão agora disponíveis no IDE. Maiores informações sobre o funcionamento e utilização do *Firebird Data Provider* podem ser encontradas na edição 66 da ClubeDelphi, ou ainda no curso de Delphi e ASP.NET da DevMedia (*http://www.devmedia.com.br/curso/ecommerce2005*).

Listagem 1. Instrução SQL do fbCommand

```
select
  E.EMP_NO,
  E.FIRST_NAME,
  E.LAST_NAME,
  D.DEPARTMENT
from
  EMPLOYEE E
inner join
  DEPARTMENT D on E.DEPT_NO = D.DEPT_NO
where upper(FIRST_NAME) like @FIRST_NAME
order by
  E.FIRST_NAME
```

Abra o editor da propriedade *Parameters* do *FbCommand* e adicione um novo parâmetro, configurando seu *ParameterName* para "@First_Name" e *SourceColumn* para "First_Name". Verifique se o *FbDbType* está como *VarChar*.

Adicione um *TextBox* e um *Button* ao formulário. Na propriedade *Text* do *Button* digite "Pesquisar". Adicione também um *DataGrid*. Clique duas vezes sobre o botão e codifique conforme a **Listagem 2**.

Listagem 2. Código do botão Pesquisar

```
FbConnection1.Open;
try
  FbCommand1.Parameters['@FIRST_NAME'].Value :=
    TextBox1.Text.ToUpper +'%';
  DataGrid1.DataSource := FbCommand1.ExecuteReader;
  DataGrid1.DataBind;
finally
  FbConnection1.Close;
end;
```

Você pode compilar a aplicação e testar informando alguma letra ou nome. Para trazer todos os registros deixe o *TextBox* em branco em clique em *Pesquisar*. Note que a cada pesquisa a página será toda atualizada (há o refresh total da página). Até agora, simplesmente montamos uma pesquisa com Firebird e ainda não estamos utilizando a tecnologia AJAX. Vamos ao próximo passo.

Utilizando o MagicAjax

Em primeiro lugar precisamos adicionar uma referência para a DLL do *MagicAjax* para que possamos usar a tecnologia. No *Project Manager* selecione o item *References* e clique com o botão direito do mouse para abrir o menu de contexto e escolha *Add Reference*. Na tela de *Add Reference*, com a tab *.NET Assemblies* selecionada, clique no botão *Browse* e encontre o arquivo *MagicAjax.dll* a partir do diretório onde você o instalou. Clique em OK para fechar o editor.

Também no *Project Manager*, clique duas vezes no arquivo *web.config* pois precisaremos editá-lo, informando à aplicação para tratar as requisições de forma diferenciada, através do *MagicAjax*. Procure pela sessão *<httpModules>* e adicione o seguinte código:

```
<add name="MagicAjax" type="MagicAjax.MagicAjaxModule, MagicAjax"/>
```

Abra o *WebForm1* e exiba seu código ASPX, pois teremos que registrar um *NameSpace* e criar um objeto *panel* do AJAX que dirá para a aplicação que todo o processamento programado nos componentes dentro desse painel será processado via AJAX, sem a necessidade de dar um *refresh* na página inteira. Para registrar informamos:

```
<%@ Register TagPrefix="ajax" Namespace="MagicAjax.UI.Controls"
  Assembly="MagicAjax" %>
```

Para criar um painel, adicione:

```
<ajax:AjaxPanel id="AjaxPanel1" runat="server">
</ajax:AjaxPanel>
```

Veja o código completo do arquivo ASPX na **Listagem 3.**

Listagem 3. Código completo do ASPX

```
<%@ Page language="c#" Debug="true"
   Codebehind="WebForm1.pas" AutoEventWireup="false"
   Inherits="WebForm1.TWebForm1" %>
<%@ Register TagPrefix="ajax" Namespace="MagicAjax.UI.Controls"
   Assembly="MagicAjax" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title></title>
  </head>
  <body>
     <form runat="server">
<ajax:AjaxPanel id="AjaxPanel1" runat="server">
  <p><font face="Verdana" size="2">
        <strong>Pesquisa com Ajax</strong>
        </font>
      </p>
      <p>
        <hr width="100%" size="1">
        <ASP:TextBox id="TextBox1" runat="server">
        </ASP:TextBox>

        <ASP:Button id="Button1" runat="server" text="Pesquisar">
        </ASP:Button></p>
        <P></P>
      <p></p>
      <p>
        <ASP:DataGrid id="DataGrid1" runat="server">
        </ASP:DataGrid>
  </p>
  </ajax:AjaxPanel>
     </form>
  </body>
</html>
```

Feito isso, você pode compilar a aplicação e fazer buscas no banco de dados. Note que ao clicar no botão, aparecerá no canto superior direito a informação *Loading* (**Figura 2**). Isso indica que a aplicação está usando a tecnologia AJAX para buscar o resultado da pesquisa. Outro fator importante é que não precisamos colocar nenhum código Java Script para utilizar a tecnologia, o que difere de outras bibliotecas que necessitam de uma codificação extra.

> **Nota:** Dependendo da quantidade de registros, a mensagem de *Loading* pode aparecer rapidamente de forma quase imperceptível.

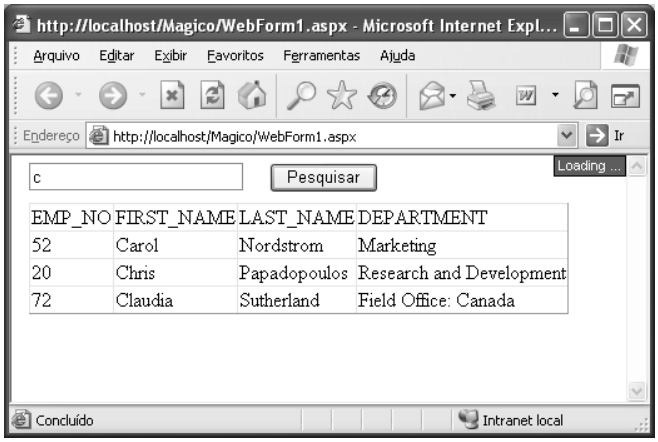


Figura 2. Aplicação com AJAX em funcionamento

Listando opções

Agora vamos criar uma nova aplicação com listas de seleções. A idéia, é primeiramente escolher um país, depois escolher um cliente daquele país para ver seus dados. Crie uma nova aplicação e dê o nome de "Clientes". Para demonstrar que a página não será totalmente recarregada, coloque um *HTML Image* da paleta *HTML Elements*. Clique com o botão direito do mouse sobre o componente e escolha *Properties*. Para *Image Source* informe: "http://www.devmedia.com.br/Imagens/logo.gif".

Logo abaixo escreva o texto "Pesquisa de Consumidores". Adicione um *HTML Horizontal Rule*, também da paleta *HTML Elements*. Abaixo, escreva o texto "País:", pressione Shift+ENTER e coloque um *DropDownList*. Pressione ENTER e escreva o texto "Consumidor:". Pressione Shift+ENTER e coloque outro *DropDownList*. Por último, adicione um *DataGrid* e configure sua aparência, clicando em *Auto Format*, ao final do *Object Inspector*. Veja como deve ficar a interface final na **Figura 3**.

Adicione um *FbConnection*, clique na propriedade *ConnectionString* e configure suas propriedade conforme a **Figura 1**. Coloque três componentes do tipo *FbCommand*, apontando suas propriedades *Connection* para o *fbConnection1* e a propriedade *CommandText* conforme a **Listagem 4.**

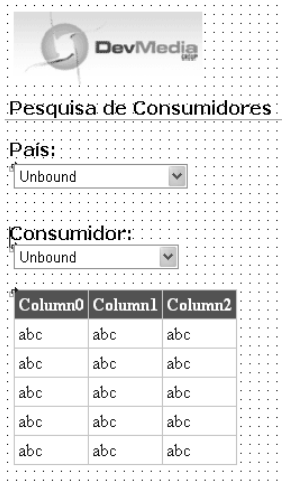


Figura 3. Aparência final da interface

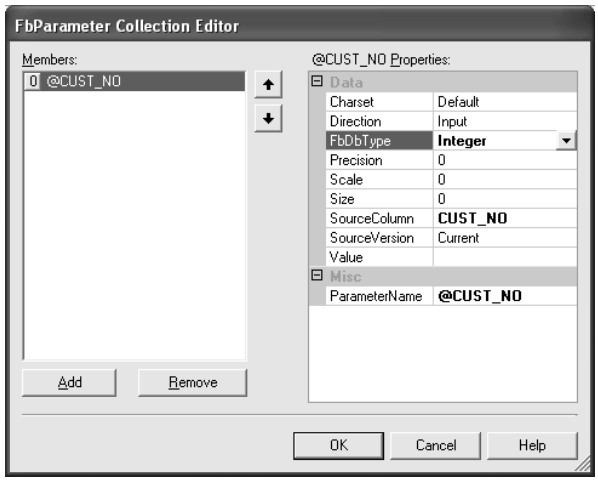


Figura 4. Configuração da propriedade Parameters do FbCommand3

Listagem 4. Código SQL dos componentes Command

Instrução SQL do FBCommand1

```
select COUNTRY
from COUNTRY
order by COUNTRY
```

Instrução SQL do FBCommand2

```
select CUST_NO, CUSTOMER
from CUSTOMER
where COUNTRY = @COUNTRY
order by CUSTOMER
```

Instrução SQL do FBCommand3

```
select CUST_NO, CUSTOMER, CONTACT_FIRST, CITY
from CUSTOMER
where CUST_NO=@CUST_NO
```

Note que no *FbCommand2* e no *FbCommand3* utilizaremos parâmetros nas instruções SQL. Então precisamos adicionar esses parâmetros na propriedade *Parameters*. Selecione o *FbCommand2* e clique na propriedade *Parameters*. No editor de parâmetros que será aberto, clique no botão *Add*, para a propriedade *ParameterName* informe "@COUNTRY" e em *SourceColumn* informe "COUNTRY".

Repita esses passos no *FbCommand3* informando *ParameterName* como "@CUST_NO" e *SourceColumn* com "CUST_NO". Como nesse casso o parâmetro será inteiro é necessário alterar a propriedade *FbDbType* para *Integer*, conforme mostra a **Figura 4**. Vamos ao código. Para o evento *Load* do *WebForm1* digite o código contido na **Listagem 5**.

Listagem 5. Código do evento Load do WebForm

```
if not Page.IsPostBack then
begin
  FbConnection1.Open;
  try
    DropDownList1.DataTextField := 'COUNTRY';
    DropDownList1.DataValueField := 'COUNTRY';
    DropDownList1.DataSource :=
      FbCommand1.ExecuteReader;
    DropDownList1.DataBind;
  finally
    FbConnection1.Close;
  end;
end;
```

Clique duas vezes sobre o *DropDownList1* e codifique-o conforme a **Listagem 6**.

Listagem 6. Código do evento SelectedIndexChanged do DropDownList1

```
FbConnection1.Open;
  try
    DropDownList2.DataTextField := 'CUSTOMER';
    DropDownList2.DataValueField := 'CUST_NO';
    FbCommand2.Parameters['@COUNTRY'].Value :=
      DropDownList1.SelectedValue;
    DropDownList2.DataSource :=
      FbCommand2.ExecuteReader;
    DropDownList2.DataBind;
  finally
    FbConnection1.Close;
  end;
```

Agora clique duas vezes no *DropDownList2* e coloque o código conforme a **Listagem 7**.

Listagem 7. Código do evento SelectedIndexChanged do DropDownList2

```
FbConnection1.Open;
  try
    FbCommand3.Parameters['@CUST_NO'].Value :=
      DropDownList2.SelectedValue;
    DataGrid1.DataSource := FbCommand3.ExecuteReader;
    DataGrid1.DataBind;
  finally
    FbConnection1.Close;
  end;
```

Para que o evento *SelectedIndexChanged* seja disparado pela aplicação, é necessário modificar a propriedade *AutoPostBack* do *DropDownList* para *True*. Faça isso nos dois componentes que foram adicionados em nossa aplicação.

Nesse momento você poderá testar a aplicação que já deve estar funcionando da forma normal, executando *Refresh* integral da página. Note que, ao selecionar algum país, o símbolo do IE (no canto superior direito) fica animado, informando que a página está sendo atualizada.

Agora é a hora de transformar a aplicação para usar o AJAX. No *Project Manager*, clique duas vezes no arquivo *web.config* para abri-lo. Procure pela sessão *<httpModules>* e adicione o código conforme o exemplo anterior (não esqueça de adicionar no *References* a DLL do AJAX).

Abra o *WebForm1* e exiba seu código ASPX para registrar a *NameSpace* do AJAX e criar um painel. Para registrar informe o seguinte código na segunda linha do arquivo:

```
<%@ Register TagPrefix="ajax"
  Namespace="MagicAjax.UI.Controls"
  Assembly="MagicAjax" %>
```

Para criar o painel, antes do primeiro *DropDownList*, digite o seguinte código:

```
<ajax:AjaxPanel id="AjaxPanel1" runat="server">
```

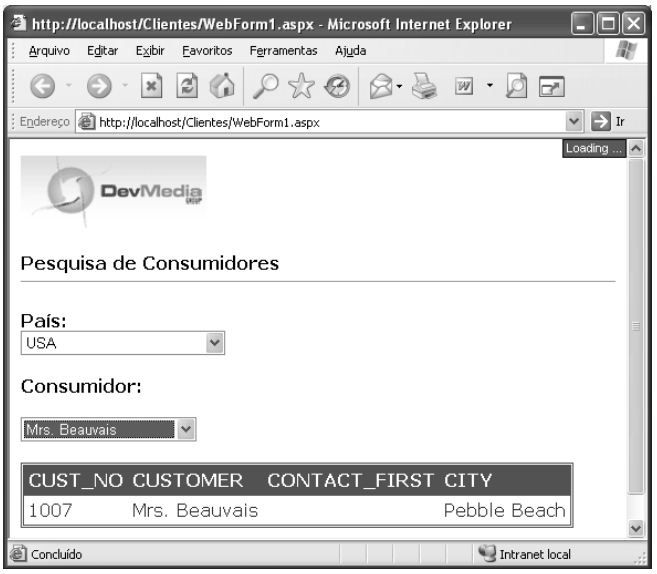


Figura 5. Aplicação com busca assíncrona dos dados, usando AJAX

Isso deve ficar antes da declaração do *DropDownList*:

```
<ASP:DropDownList id="DropDownList1" runat="server"
width="185px" autopostback="True"></ASP:DropDownList>
```

Para finalizar o painel, antes da linha: *</form>*, digite: "</ajax:AjaxPanel>" (assim fechamos o painel depois do *DataGrid*). Pronto, com isso você poderá testar a aplicação e ver que as informações da segunda lista de opções são buscadas através do AJAX, assim como a grade de informações do cliente/consumidor escolhido (**Figura 5**).

Isso pode ser confirmado, pois o símbolo do IE (no canto superior direito) não fica animado quando um país ou consumidor é selecionado.

Conclusão

O *MagicAjax* facilita em muito a criação de aplicações ASP.NET que usam a tecnologia AJAX. Essa facilidade irá difundir esse tipo de aplicação que economiza tráfego de informações entre clientes e servidores. Seja inovador, comece a utilizar essa tecnologia em suas aplicações. ■

Links

Treinamento on-line exclusivo em Delphi e ASP.NET da DevMedia
www.devmedia.com.br/curso/ecommerce2005/

O que é precioso você não entrega nas mãos de qualquer um. Por isso, quando pensar em Internet Data Center, procure a líder em número de clientes no Rio. Procure a ALOG. Utilizamos as mesmas estruturas da antiga filial carioca da .comDominio, o que lhe garante uma superestrutura em um prédio-cofre com sistemas de controle de acesso, de detecção e combate a incêndio com gás FM200, ambiente climatizado, no breaks e geração própria de energia de 1200KVA's, infra-estrutura de redes e acesso redundantes com monitoramento 24x7. Tudo para atender aos sistemas de missão crítica dos nossos clientes. Mas não é só isso. A ALOG ainda dispõe de atendimento personalizado, eficiente e ágil, com capacidade para atender aos nossos mais de 240 clientes corporativos com rapidez.
Saiba tudo o que podemos fazer pela sua empresa. Ligue para nós, (21) 3083-3333.

ALOG, SEUS DADOS BEM-CUIDADOS.

Controle de incêndio com FM200

1200KVA's de geração própria

Controle de acesso

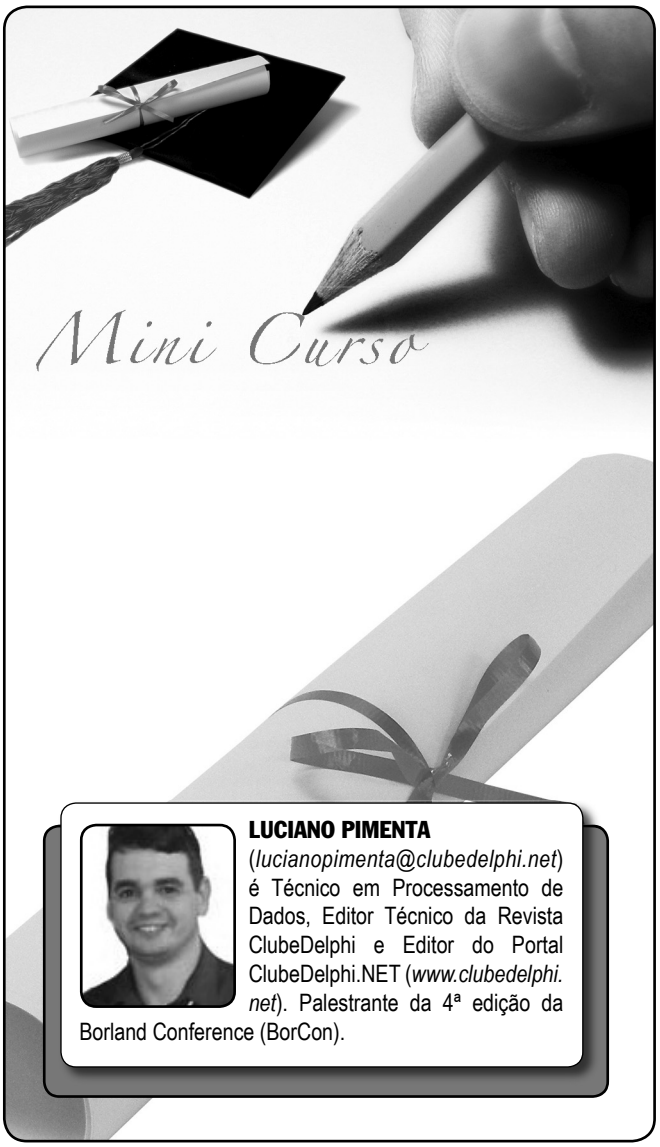
Portas blindadas

ALOG
data centers

www.alog.com.br

Aplicação Web Completa

Cadastros, consultas e customização de controles - Parte 2



LUCIANO PIMENTA (*lucianopimenta@clubedelphi.net*) é Técnico em Processamento de Dados, Editor Técnico da Revista ClubeDelphi e Editor do Portal ClubeDelphi.NET (*www.clubedelphi.net*). Palestrante da 4ª edição da Borland Conference (BorCon).

Começamos na edição anterior nosso mini-curso de uma aplicação Web completa. Veremos neste artigo como criar os cadastros de produtos. Faremos uma pesquisa para o usuário localizar os produtos que deseja, além de várias configurações de componentes e boas práticas de programação Web.

Cadastro de produtos

Como estamos trabalhando em uma aplicação Web, como comentado no artigo anterior, temos sempre que primar pela performance da aplicação, retornando do banco de dados apenas a quantidade mínima de registros. Assim como em aplicações Desktop, no cadastro de produtos não teremos todos os registros da tabela e uma barra de navegação, mas sim apenas o registro escolhido pelo usuário na consulta que criaremos mais adiante.

Crie um novo *WebForm* (*File>New>Other>New ASP.NET Files>ASP.NET Pages*), renomei-o para "cadastroprodutos.aspx" e adicione os dois *User Controls* criados na edição anterior. Na **Tabela 1** temos a nomenclatura, tipo e valores de propriedades alterados para os componentes que adicionaremos no formulário (com exceção dos *Labels*). Nosso formulário ficará semelhante ao da **Figura 1.**

Clique com o botão direito no *File Upload* e escolha a opção *Run As Server Control*, assim podemos referenciar o mesmo no código. No *Click* do botão *Cancelar* vamos apenas redirecionar para a página principal, com o seguinte código:

```
Response.Redirect('Home.aspx');
```

Stored Procedure de inserção e atualização

Vamos agora criar uma Stored Procedure para a inserção dos dados no banco. Execute o código da **Listagem 1** no IBExpert ou outra ferramenta que esteja usando para manutenção do banco de dados da aplicação.

Componente	Nome	Propriedade/Valor
TextBox	txtProduto	
DropDownList	dpCategoria	
TextBox	txtPreco	
TextBox	txtDescricao	TextMode=MultiLine
HTML File Upload	File1	
Image	imgFigura	
Button	btnSalvar	
Button	btnCancelar	

Tabela 1. Componentes adicionados no formulário

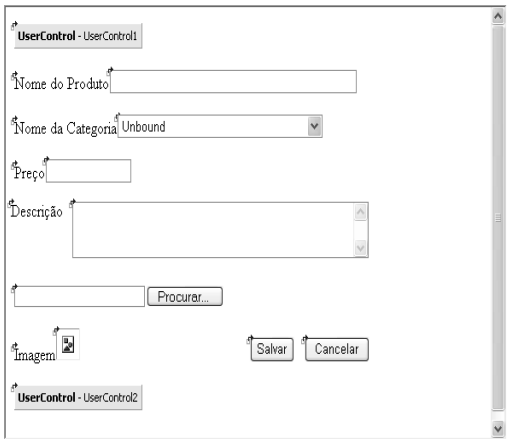


Figura 1. Tela de cadastro de produtos

Listagem 1. Stored Procedure de inserção/atualização de produtos

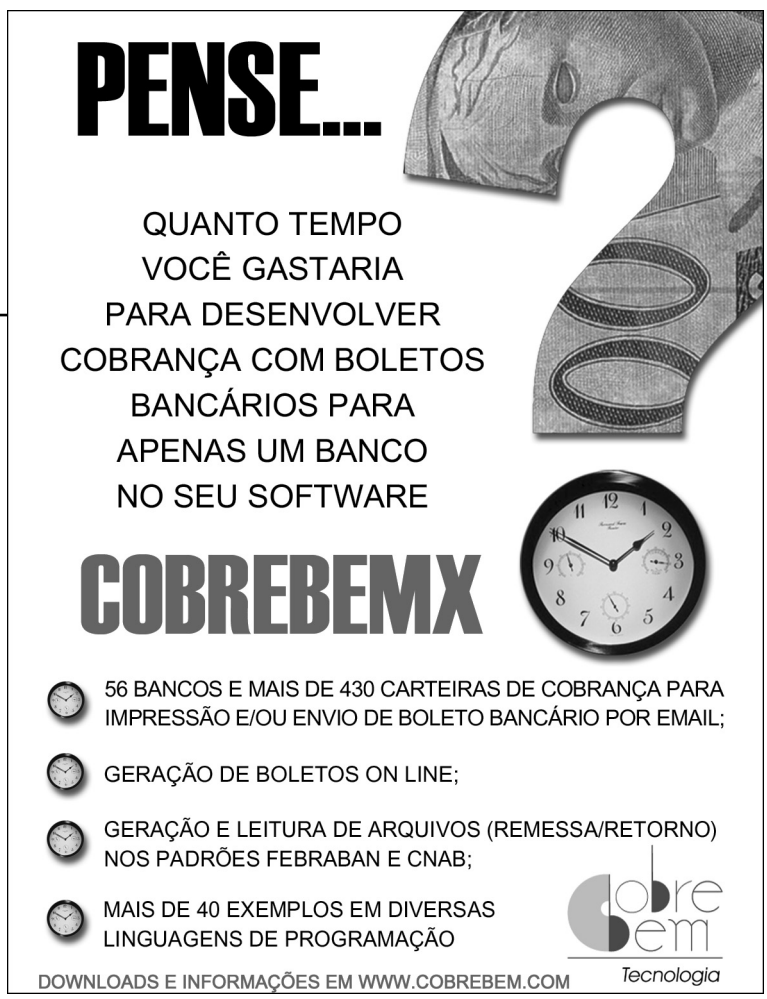
```
CREATE PROCEDURE PRODUTOS_INS_UPD (
  ID_PRODUTO INTEGER,
  ID_CATEGORIA INTEGER,
  NOME_PRODUTO VARCHAR(50),
  PRECO NUMERIC(9,2),
  DESCRICAO VARCHAR(50),
  URL VARCHAR(50))
AS
BEGIN
  IF (EXISTS(SELECT ID_PRODUTO FROM PRODUTOS WHERE (
    ID_PRODUTO = :ID_PRODUTO))) THEN
    UPDATE PRODUTOS
    SET ID_CATEGORIA = :ID_CATEGORIA,
        NOME_PRODUTO = :NOME_PRODUTO,
        PRECO = :PRECO,
        DESCRICAO = :DESCRICAO,
        URL = :URL
    WHERE (ID_PRODUTO = :ID_PRODUTO);
  ELSE
    INSERT INTO PRODUTOS (
        ID_PRODUTO,
        ID_CATEGORIA,
        NOME_PRODUTO,
        PRECO,
        DESCRICAO,
        URL)
    VALUES (
        GEN_ID(GEN_CATEGORIA_ID, 1),
        :ID_CATEGORIA,
        :NOME_PRODUTO,
        :PRECO,
        :DESCRICAO,
        :URL);
END
```

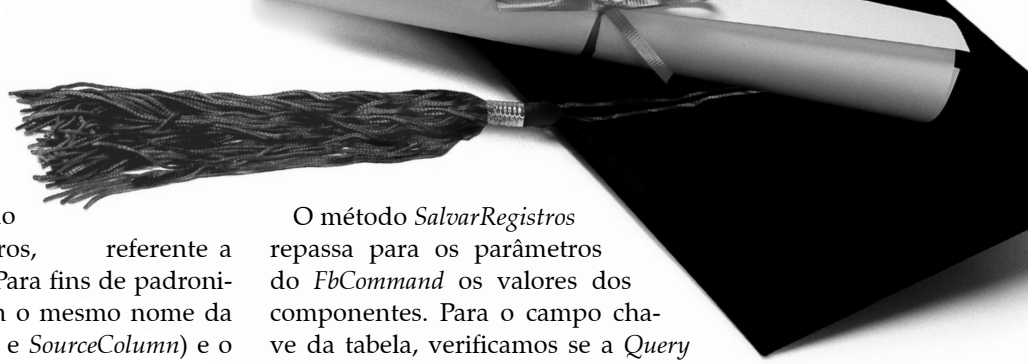
Não estranhe o código da listagem anterior, pois nossa SP possui duas funcionalidades: inserção ou atualização. A SP verifica se o valor do parâmetro ID_PRODUTO existe (*if*), e se a condição for verdadeira, então quer dizer que estamos atualizando os dados da tabela e executamos o comando *Update*. Caso a condição seja falsa, indica que estamos inserindo dados na tabela, assim executamos o comando necessário para inserção dos dados. Note o *Generator* nos parâmetros de inserção. Com isso, usamos apenas uma SP para realizar inserção ou atualização dos produtos, otimizando nosso código.

> **Nota**: É altamente recomendado o uso de Stored Procedures quando se usa o ADO.NET e ASP.NET. Esse é um fator crítico em ambiente Web, para garantir a performance.

Voltando ao formulário de cadastro de produtos, adicione um *FbConnection* e um *FbCommand*. Faça acesso ao banco de dados da aplicação e configure a ligação dos componentes. Na propriedade *CommandText* adicione o seguinte código:

```
execute procedure PRODUTOS_INS_UPD (?, ?, ?, ?, ?, ?)
```





Acesse a propriedade *Parameters* do componente e adicione seis parâmetros, referente a cada campo da tabela PRODUTOS. Para fins de padronização, dê o nome ao parâmetro com o mesmo nome da coluna (propriedades *ParameterName* e *SourceColumn*) e o tipo (propriedade *FbDbType*) seja o mesmo tipo da coluna cadastrada na tabela do banco.

No editor de códigos vamos criar dois métodos: um para salvar o registro e outro para salvar a figura do produto em uma pasta no servidor. Adicione os seguintes métodos na seção *public* da unit:

```
public
  { Public Declarations }
  procedure SalvarRegistro;
  function SalvarFiguraDisco: string;
```

Adicione o código para a implementação, conforme a **Listagem 2**.

Listagem 2. Métodos para salvar a figura em disco e o registro no banco

```
function TWebForm1.SalvarFiguraDisco: string;
var
  aFile, aDiretorio, aPath: string;
begin
  { Declare em uses System.IO }
  aFile :=  Path.GetFileName(
    file1.PostedFile.FileName);
  aDiretorio := 'http://localhost/CursoASPNET/images;
  aPath := aDiretorio + '/' + aFile;
  file1.PostedFile.SaveAs(Server.MapPath(
    '.\images\' + aFile));
  imgFigura.ImageUrl := aPath;
  Result := aPath;
end;

procedure TWebForm1.SalvarRegistro;
begin
  FbConnection1.Open;
  try
    if file1.PostedFile.FileName <> '' then
    begin
      with FbCommand1 do
      begin
        if Request.QueryString[
          'CodProduto'] <> nil then
          Parameters[0].Value :=
            Request.QueryString['CodProduto']
        else
          Parameters[0].Value := &Object(0);
        Parameters[1].Value :=
          dpCategoria.SelectedValue;
        Parameters[2].Value := txtProduto.Text;
        Parameters[3].Value := txtPreco.Text;
        Parameters[4].Value := txtDescricao.Text;
        Parameters[5].Value := SalvarFiguraDisco;
        ExecuteNonQuery;
        Response.Write('<script>Javascript:alert('+
          ''Cadastro efetuado com sucesso!!'');'+
          '</script>');
      end;
    end
    else
      Response.Write('<script>Javascript:alert('+
        ''Escolha um arquivo para salvar o '+
        'registro!'');</script>');
  finally
    FbConnection1.Close;
  end;
end;
```

A função *SalvarFiguraDisco* retorna o caminho da figura que estamos salvando em disco. Utilizamos essa técnica pois será mais fácil referenciar a figura nos demais módulos do site.

Nota: A pasta *images* foi criada na edição anterior.

O método *SalvarRegistros* repassa para os parâmetros do *FbCommand* os valores dos componentes. Para o campo chave da tabela, verificamos se a *QueryString CodProduto* está vazia, isso é necessário para saber se estamos trabalhando com edição ou inserção.

Vamos agora carregar os dados da tabela de categorias no *DropDownList*. Adicione um *FbCommand*, faça a ligação do *FbConnection* e na propriedade *CommandText* digite:

```
select ID_CATEGORIA, NOME_CATEGORIA
from CATEGORIA
```

Crie um novo método, chamado "CarregaCategoria" e implemente-o conforme o código da **Listagem 3**.

Listagem 3. Código para carregar o DropDownList de categorias

```
procedure TWebForm1.CarregaCategoria;
begin
  FbConnection1.Open;
  try
    dpCategoria.DataSource :=
      FbCommand2.ExecuteReader;
    dpCategoria.DataTextField := 'NOME_CATEGORIA';
    dpCategoria.DataValueField := 'ID_CATEGORIA';
    dpCategoria.DataBind;
  finally
    FbConnection1.Close;
  end;
end;
```

No evento *Load* da página adicione o seguinte código:

```
if not IsPostBack then
  CarregaCategoria;
```

Por fim, no *Click* do botão *Salvar*, basta chamar o método *SalvarRegistro*. Antes de testar, adicione diretamente no banco de dados alguns valores para a tabela *Categorias*. Caso deseje, faça uma página semelhante para o cadastro das categorias.

No design da página, clique com o botão direito do mouse e escolha a opção *View In Browser*. Após a compilação clique em OK e cadastre os produtos normalmente. Uma dica é adicionar as validações para o cadastro, semelhante ao que fizemos com o cadastro de usuários, na edição anterior. Veja na **Figura 2** a aplicação em execução.

Uma dica para o leitor implementar é a atualização do produto. Para isso, basta passar o código do produto como parâmetro na URL, selecionar os dados e exibir os mesmos nos controles, como por exemplo:

```
{ Evento Load da página }
if Request.QueryString['CodProduto'] <> nil then
begin
  SelecionaDados;
  ExibeControles;
end;
```

E não precisamos alterar mais nada, pois no código que salva os dados, verificamos se o valor do *QueryString* está vazio, assim passamos o valor zero (na SP será feita a inclusão). Como preenchemos o *QueryString* com o código

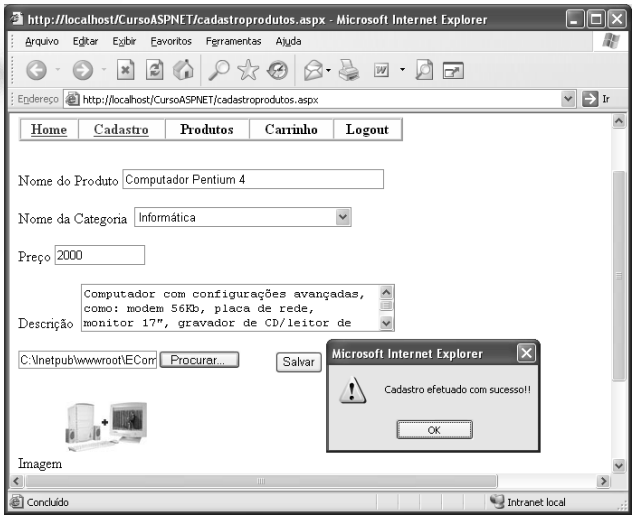


Figura 2. Cadastro de Produtos

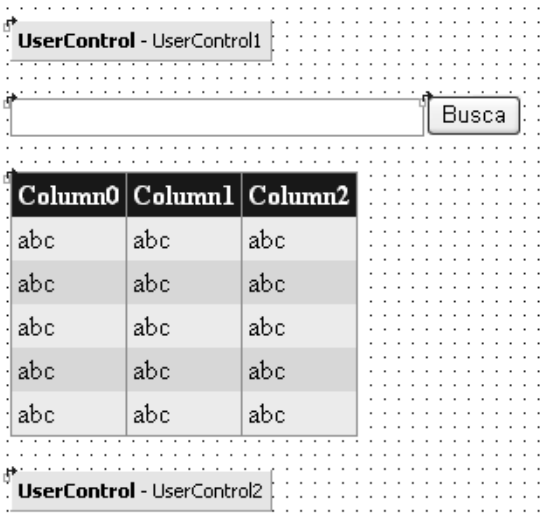


Figura 3. Formulário de busca da aplicação

do produto para atualização, o mesmo será usado no parâmetro do *FbCommand* (executando o comando *Update* da Stored Procedure no banco).

Consulta de produtos

Para visualizar os produtos do site, temos que ter uma consulta para o usuário escolher aquele que deseja obter mais informações e/ou comprar. Essa funcionalidade é bastante usada em sites de e-commerce. Crie um novo *WebForm* (*File>New>Other>New ASP.NET Files>ASP.NET Pages*), renomei-o para "consultaprodutos.aspx" e adicione os dois *User Controls* criados na edição anterior.

Adicione um *TextBox* ("txtBusca"), um *Button* ("btnBusca") e um *DataGrid*. Adicione também os componentes de acesso a dados (*FbConnection* e *FbCommand*). Faça a ligação entre os componentes, formate o *DataGrid* através das opções de formatação (*Auto Format*). Seu formulário de busca deve estar semelhante a **Figura 3**.

Na propriedade *CommandText* do *FbCommand* digite o seguinte código:

```
select ID_PRODUTO, NOME_PRODUTO, URL
from PRODUTOS
where UPPER(NOME_PRODUTO) like ?
```

Adicione um parâmetro no *FbCommand*, semelhante ao que já fizemos no cadastro anterior e digite o código da **Listagem 4**, no *Click* do botão.

Listagem 4. Consulta de produtos no banco

```
FbConnection1.Open;
try
  FbCommand1.Parameters[0].Value := txtBusca.Text.ToUpper+ '%';
  DataGrid1.DataSource := FbCommand1.ExecuteReader;
  DataGrid1.DataBind;
finally
    FbConnection1.Close;
end;
```

Note que chamamos *ToUpper* da propriedade *Text* do *TextBox* para que o valor seja repassado todo em maiúsculo. Passamos o parâmetro para o *FbCommand* e executamos a consulta. Veja que colocamos o caractere coringa (%) apenas no final do parâmetro, assim se o usuário digitar "a", serão mostrados os produtos que começam com a referida letra.

Alguns podem querer que a pesquisa seja realizada por qualquer parte do nome do produto, adicionando assim o coringa também, antes do parâmetro. Esse tipo de pesquisa não traz boa performance para a aplicação, pois a quantidade de registros retornados pode ser muito grande.

Outra dica é solicitar ao usuário que digite uma quantidade mínima de caracteres para realizar a busca. Por exemplo, com o seguinte código, você pode forçar o usuário a digitar no mínimo três letras para realizar a busca:

```
if txtBusca.Text.Length > 3 then
  { Executa a consulta }
else
  { Aviso de quantidade mínima }
```

Para testar, abra a página usando a opção *View In Browser*.

Personalizando o DataGrid

Sem dúvida nenhuma o *DataGrid* é um dos componentes mais usados em aplicações Web, pois mostra os dados de maneira tabular, de forma rápida e simples. Podemos ter várias configurações do *DataGrid* e uma que faremos agora é mostrar uma figura.

Abra o editor do *DataGrid* através da propriedade *Columns* e desmarque a opção de gerar automaticamente as colunas (*Create columns automatically at run time*). Adicione respectivamente, um *Hyperlink Column* e dois *Bound Column*.

No *Hyperlink* configure as seguintes propriedades: em *Header Text* digite "Código" e em *Text Field* digite "ID_PRODUTO". Para as outras duas colunas configure *Header Text* como "Nome do Produto" e "Figura" e *Data Field* com os campos NOME_PRODUTO e URL.

Na propriedade *Data Formatting string* da coluna URL, adicione o seguinte valor: "<img src={0}>". Assim, no momento da renderização do controle, será adicionado no

HTML final, a tag *img src* e o valor do campo URL, que é o local onde esta salvo as imagens.

Mostraremos assim, no *DataGrid*, imagens que armazenamos no disco (**Figura 4**), poupando espaço no banco e codificação para a exibição da figura, sem falar na performance que é muito maior.

Vale ressaltar que o tamanho da imagem, conta muito, pois imagens grandes podem distorcer e deixar o site com um layout estranho. Por isso, vale a dica de padronizar o tamanho das imagens, claro.

Quando o usuário realizar a busca, temos que dar a opção do mesmo abrir as informações do produto que ele deseja comprar. Assim na coluna ID_PRODUTO, vamos colocar um valor para redirecionarmos o usuário para uma página com opção de compra do produto.

Na propriedade *URL field* digite "ID_PRODUTO" e em *URL format string*, digite: "visualisaproduto.aspx?CodProduto={0}". A página *visualisaproduto.aspx*, ainda não está criada, faremos isso agora.

Comprando o produto

Crie um novo *WebForm*, conforme a técnica anterior, dando o nome de "visualisaproduto.aspx". Mostraremos as informações do produto, dando a possibilidade de compra do mesmo. Nessa página é onde podemos colocar todas as informações/detalhes do produto, para que o usuário saiba suas características.

Uma dica é adicionar comentários nessa página, sobre o produto de quem já realizou a compra do mesmo. Adicione os *User Controls* no formulário.

> **Nota:** Como já realizamos o cadastro do produto, essa página destina-se apenas a exibir as informações, então não precisamos usar *TextBox* para todos os campos, por exemplo, e tão pouco um *Drop DowList*, pois não alteraremos a categoria do produto.

Veja na **Figura 5**, como ficaram os componentes no formulário (perceba o nome dos mesmos).

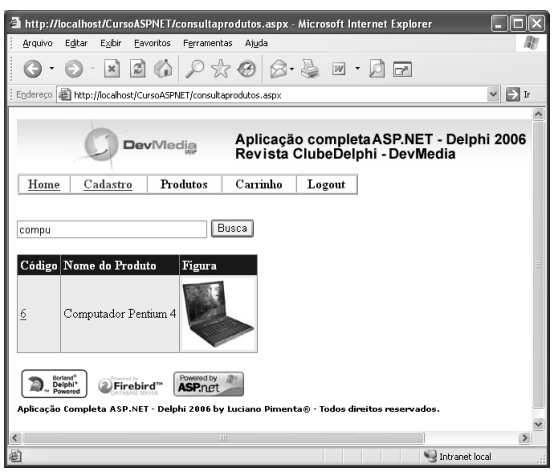


Figura 4. Mostrando imagens no DataGrid

Temos um *ImageButton* que possui a figura, mostrando ao usuário onde clicar para que o mesmo possa adicionar ao carrinho de compras o produto. No *TextBox* altere a propriedade *ReadOnly* para *True* e *TextMode* para *MultiLine*.

Adicione os componentes de acesso a dados e vamos usar a consulta que está na **Listagem 5**, que deve ser adicionada na propriedade *CommandText* do *FbCommand*:

Listagem 5. Consulta para mostrar os dados do produto

```
select PRODUTOS.ID_PRODUTO, PRODUTOS.NOME_PRODUTO,
  CATEGORIA.NOME_CATEGORIA, PRODUTOS.PRECO,
  PRODUTOS.DESCRICAO, PRODUTOS.URL
from PRODUTOS
  inner join CATEGORIA on (PRODUTOS.ID_CATEGORIA =
  CATEGORIA.ID_CATEGORIA)
where PRODUTOS.ID_PRODUTO=?
```

Veja que na consulta estamos trazendo a categoria do produto e parametrizando a mesma pelo código do produto. Adicione o parâmetro no *FbCommand* (com o mesmo nome do campo). Adicione um método chamado "Select" no código e implemente-o conforme a **Listagem 6**.

Listagem 6. Código que mostra os dados da consulta na tela

```
procedure TWebForm1.Select;
var
  dr: FbDataReader;
begin
  if Request.QueryString['CodProduto'] <> nil then
  begin
    FbConnection1.Open;
    try
      FbCommand1.Parameters[0].Value :=
        Request.QueryString['CodProduto'];
      dr := FbCommand1.ExecuteReader;
      if dr.HasRows then
      begin
        dr.Read;
        lblProduto.Text := dr['NOME_PRODUTO'].ToString;
        lblCategoria.Text := dr['NOME_CATEGORIA'].ToString;
        lblPreco.Text :=  System.&String.Format('{0:c}', dr['PRECO']);
        txtDescricao.Text := dr['DESCRICAO'].ToString;
        Image1.ImageUrl := dr['URL'].ToString;
      end;
    finally
      FbConnection1.Close;
    end;
  end;
end;
```

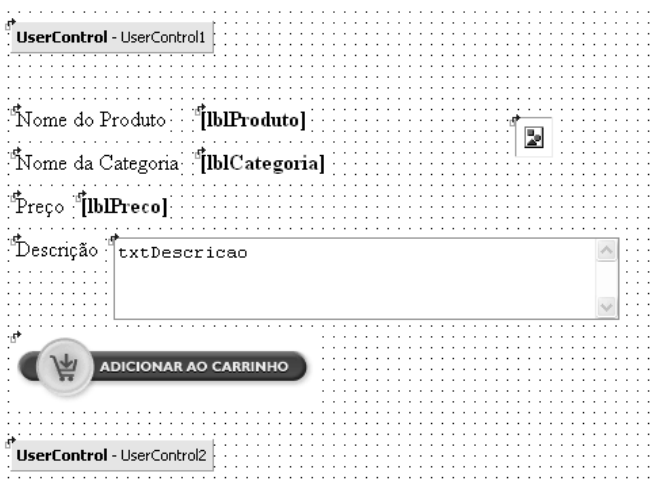


Figura 5. Componentes da página para visualizar os dados do produto e opção de compra

No código anterior, temos algo diferente: *System.&String.Format('{0:c}', dr['PRECO'])*. Esse código formata o valor do campo PRECO para o formato moeda. Com esse código podemos fazer várias formatações com data/hora, casas decimais etc. Para finalizar, no *Load*, digite:

```
if not IsPostBack then
  Select;
```

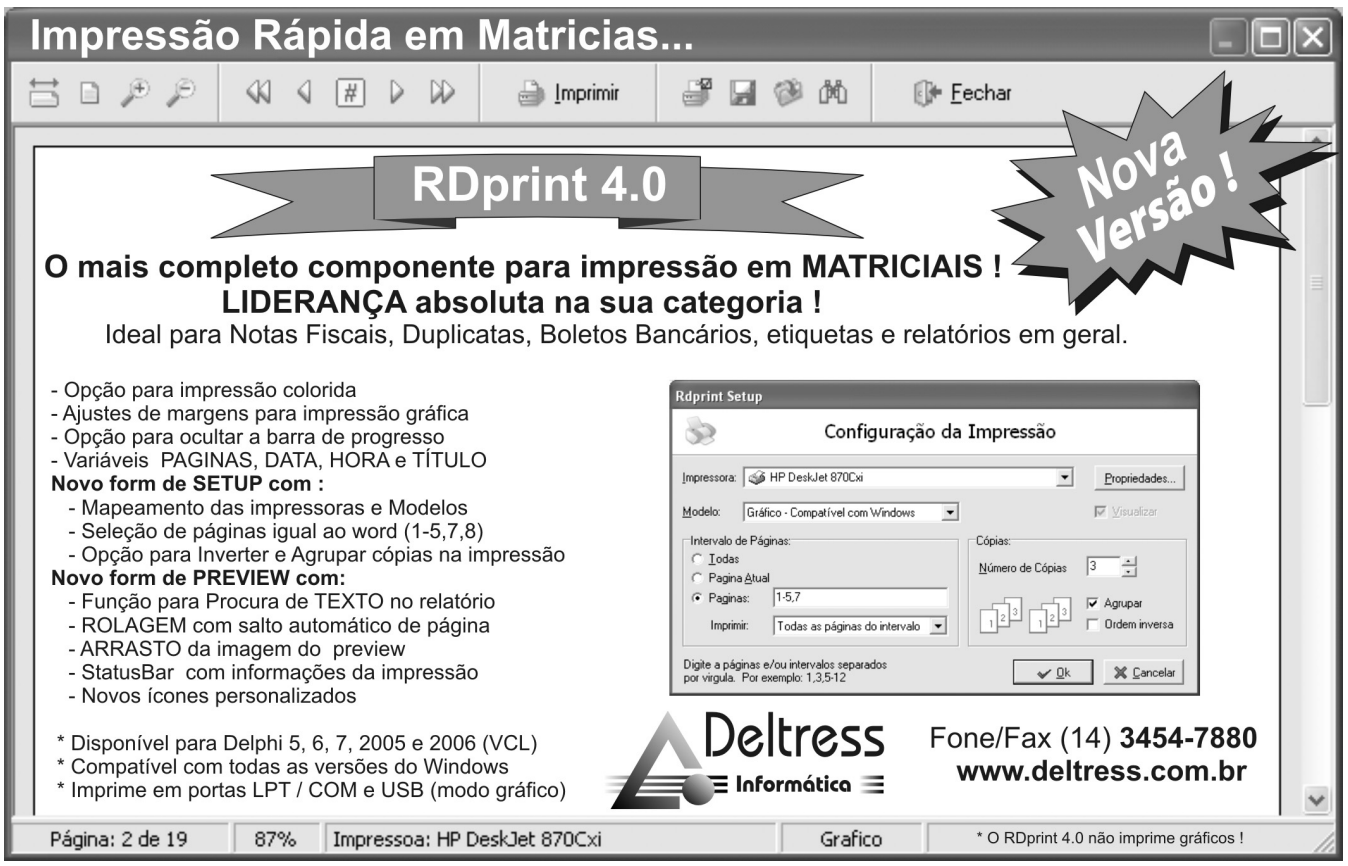
Abra o *header.ascx* e no componente *Hyperlink* chamado *Produtos*, adicione na propriedade *NavigateURL*, o valor: "consultaprodutos.aspx". Rode a aplicação, abra a consulta de produtos. Escolha o produto no link do *DataGrid* e visualize os dados do produto na página recém criada.

"Data Modules" no ASP.NET

Vamos implementar na página principal da aplicação uma filtragem de produtos, para que o usuário possa navegar no site através das categorias do mesmo. Assim, precisamos abrir o *Home.aspx* e adicionar um *ListBox* ("lstCategoria") na página.

Temos que preencher o *ListBox* com o nome das categorias cadastradas no banco, semelhante ao que fizemos com o *DropDownList* do cadastro de produtos. Mas será que não podemos aproveitar essa consulta? Sim, basta organizarmos o código em units.

Estamos acostumados a trabalhar com DataModules em aplicações Win32, onde podemos concentrar os componentes de acesso a dados e regras de negócios. Mas posso fazer isso em ASP.NET? Sim, podemos "simular" um DataModule no ASP.NET. Crie uma nova classe (*File>New>Other>New Files>Class*), salve e dê o nome de "uDM.pas". O código da



classe deve ficar, semelhante a **Listagem 7**.

Listagem 7. Simulando um DataModule em ASP.NET

```
unit uDM;

interface

uses System.ComponentModel;

type
  TDM = class (System.ComponentModel.Component)
  private
    { Private Declarations }
  public
    constructor Create;
  end;

implementation

constructor TDM.Create;
begin
  inherited Create;
  // TODO: Add any constructor code here
end;
end.
```

Feche a classe e abra-a novamente. Note que agora temos um container onde podemos colocar nossos componentes de acesso e dados e na classe trabalhar normalmente com código para as regras de negócio.

Adicione um *FbConnection* no "DataModule" e faça a configuração com o banco. Veja que o Delphi cria o *InitializeComponent* automaticamente, assim precisamos apenas chamar o *InitializeComponent* dentro do *Create* da classe.

Na seção *public*, crie uma função que retornará um *FbDataReader*, chamada "GetCategorias" e implemente-a conforme a **Listagem 8**.

Listagem 8. Função para retornar as Categorias cadastradas no banco

```
function TDM.GetCategorias: FbDataReader;
var
  dr: FbDataReader;
  cmd: FbCommand;
begin
  { Declare no uses System.Data }
  FbConnection1.Open;
  cmd := FbCommand.Create('select * from CATEGORIA', FbConnection1);
  dr := cmd.ExecuteReader(CommandBehavior.CloseConnection);
  Result := dr;
end;
```

O código da listagem anterior cria uma instância de *FbCommand* e passa como parâmetro a instrução SQL que usaremos, juntamente com a conexão do banco (*FbConnection1*). Após chamamos o *ExecuteReader* passando como parâmetro o *CommandBehavior.CloseConnection* que ficará encarregado de liberar a conexão com o banco. Assim não precisamos fechar a conexão nesse código, ela será fechada quando chamarmos o método *Close* do *DataReader*.

Nota: Para saber mais sobre as opções do *CommandBehavior*, dê uma olhada na documentação do .NET Framework.

Para testar, abra o cadastro de produtos e altere o código do *CarregaCategoria* para o código da **Listagem 9**.

Listagem 9. Alterando o código que retorna as categorias

```
var
  DM: TDM;
  dr: FbDataReader;
begin
  { Declare em uses uDM e FirebirdSql.Data.Firebird }
  DM := TDM.Create;
  dr := DM.GetCategorias
  dpCategoria.DataSource := dr;
  ...
  dr.Close;
end;
```

No *Home.aspx*, vamos implementar um código igual ao *CarregaCategoria* do cadastro de produtos, apenas alterando o nome do componente (que nesse caso é *lstCategoria*). Não esqueça de fazer a chamada ao método no *Page_Load* do formulário. Para finalizar, adicione um *DataGrid* e implemente o código da **Listagem 10** no "DataModule".

Listagem 10. Filtra os produtos de acordo com a categoria

```
function TDM.GetProdutos(
  aIdCategoria: string): FbDataReader;
var
  dr: FbDataReader;
  cmd: FbCommand;
begin
  FbConnection1.Open;
  cmd := FbCommand.Create(
    'select ID_PRODUTO, NOME_PRODUTO, URL '+
    'from PRODUTOS where ID_CATEGORIA=?',
    FbConnection1);
  cmd.Parameters.Add('ID_CATEGORIA',
    FbDbType.Integer, 0,
    'ID_CATEGORIA').Value := aIdCategoria;
  dr := cmd.ExecuteReader(
    CommandBehavior.CloseConnection);
  Result := dr;
end;
```

A diferença desse código fica por conta que estamos criando um parâmetro em tempo de execução. Volte a página *Home.aspx* e altere a propriedade *AutoPostBack* do *lstCategoria* para *True*.

Isso indica que ao clicarmos no item do *lstCategoria*, o evento *SelectedIndexChanged* será disparado. No referido evento adicione o código da **Listagem 11.**

Listagem 11. Código para filtrar os dados no DataGrid ao clicar no lstCategoria

```
var
  DM: TDM;
  dr: FbDataReader;
begin
  DM := TDM.Create;
  dr := DM.GetProdutos(lstCategoria.SelectedValue);
  DataGrid1.DataSource := dr;
  DataGrid1.DataBind;
  dr.Close;
end;
```

Para finalizar, faça a mesma técnica de formatação do *DataGrid*, utilizada no formulário de pesquisa, para mostrar a figura e também redirecionar o usuário para a página de compra do produto, ao clicar no *DataGrid*.

Uma dica é copiar o *DataGrid* da página de consulta e colar na *Home.aspx*, apenas alterando sua formatação (*Auto Format*), caso seja necessário. Veja na **Figura 6** a aplicação em execução.

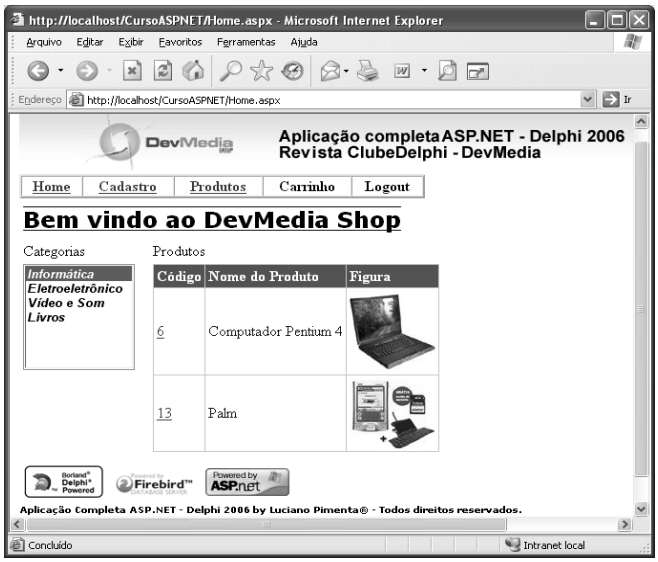


Figura 6. Filtrando os produtos por categoria

Mensagens no ASP.NET

Estamos acostumados a usar mensagens de alerta e confirmação em aplicações Win32. Para trabalhar com esse tipo de mensagens no ASP.NET devemos usar Java Script, como fizemos anteriormente. Mas e se tivéssemos um componente que nos tirasse esse trabalho?

Pois bem, Rodrigo Glauser, criou um componente *free*, onde podemos usar em nossa aplicação, que encapsula *script client side*, interagindo com a aplicação ASP.NET. Para saber como instalar o componente e baixar o mesmo, veja uma vídeo aula que criei no seguinte link: *www.devmedia.com.br/visualizacomponente.aspx?comp=1405&site=3*.

> **Nota:** Junto ao download dos arquivos do artigo, encontra-se o componente.

A instalação é rápida e simples. Mostrarei aqui, como adaptar os exemplos mostrados até agora, utilizando o componente. Por exemplo, nos cadastros, podemos adicionar o componente no formulário e alterar o código (de cada cadastro).

Como também no formulário de consulta, onde mostramos uma mensagem ao usuário que digitar menos de três caracteres na busca, conforme a **Listagem 12**.

Listagem 12. Alterando o código para utilizar o componente de mensagens

```
Cadastro de Usuários e Produtos
...
    ExecuteNonQuery;
    MessageBox1.ShowMessage(
      'Cadastro efetuado com sucesso!');
  end;

  finnally
    FbConnection1.Close;
  end;

  except
    MessageBox1.ShowMessage('Erro!');
  end;

Consulta de Produtos
...
   DataGrid1.DataBind;
 end

 else
   MessageBox1.ShowMessage('Digite mais de 3 digitos!');
```

Usamos apenas mensagens de alerta, mas podemos usar mensagens de confirmação, onde o usuário escolherá opções (Sim ou Não, por exemplo), que veremos como funciona nos próximos artigos e exemplos.

Conclusão

Nessa segunda parte do curso, realizamos o cadastro e consulta de produtos. Configuramos o *DataGrid* para mostrar figuras e que usasse um link para passar como parâmetro o código do produto e redirecionar para a página de compra. Criamos na página principal, uma opção de filtragem dos produtos por categoria.

Vimos como simular um DataModule em aplicações ASP.NET, seguindo o mesmo padrão que estamos acostumados a trabalhar na plataforma Win32. Por fim, mostramos um ótimo componente para exibir caixas de mensagens e confirmação, sem a utilização direta de Java Script (o mesmo esta encapsulado no componente).

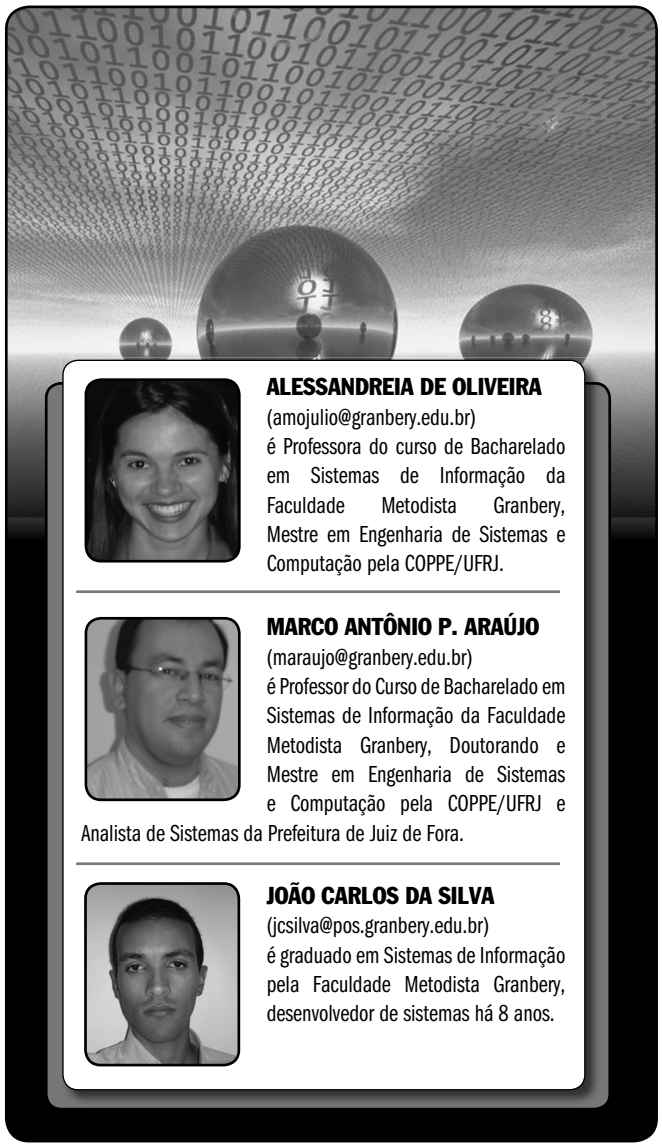
No próximo artigo, faremos uma das partes mais importantes do nosso mini-curso, o carrinho de compras. Um grande abraço a todos e até lá! ■

+ de 80.000 membros cadastrados
+ de 15.000 exemplos com fontes
+ de 900 apostilas
+ de 4.000 dicas
Fórum Delphi
Artigos
TOTALMENTE GRÁTIS
www.delphi.eti.br
Um dos maiores sites de apoio a desenvolvedores Delphi do Brasil!!!

JOÃO CARLOS DA SILVA - ALESSANDREIA DE OLIVEIRA - MARCO ARAÚJO

POO no Delphi

Conceitos e Implementação - Parte 2



ALESSANDREIA DE OLIVEIRA
(amojulio@granbery.edu.br)
é Professora do curso de Bacharelado em Sistemas de Informação da Faculdade Metodista Granbery, Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ.

MARCO ANTÔNIO P. ARAÚJO
(maraujo@granbery.edu.br)
é Professor do Curso de Bacharelado em Sistemas de Informação da Faculdade Metodista Granbery, Doutorando e Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ e Analista de Sistemas da Prefeitura de Juiz de Fora.

JOÃO CARLOS DA SILVA
(jcsilva@pos.granbery.edu.br)
é graduado em Sistemas de Informação pela Faculdade Metodista Granbery, desenvolvedor de sistemas há 8 anos.

No artigo da edição anterior iniciamos a apresentação de alguns dos principais conceitos da programação orientada a objetos (POO) no Delphi, abordando classes, atributos, métodos, encapsulamento e herança. Estes conceitos foram apresentados através de exemplos a partir de um modelo de classes representando departamentos e seus funcionários, de diferentes tipos. Este artigo complementa o anterior abordando outros importantes conceitos de POO como polimorfismo, associação e interfaces, além de apresentar um estudo de caso que se utiliza dos conceitos apresentados em ambos os artigos, mostrando como podem ser utilizados na prática.

Polimorfismo

O polimorfismo é uma característica da Orientação a Objetos que permite aos objetos das classes terem comportamentos diferentes, sejam de seus ancestrais, da própria classe ou mesmo de outras classes. Assim, os métodos definidos e implementados em classes ancestrais, podem ser redefinidos e reimplementados em classes descendentes.

Para que isso funcione adequadamente, é necessário garantir qual implementação de um método será executada, visto que pode existir mais de uma com o mesmo nome, seja na hierarquia ou dentro da própria classe. Dessa forma o Delphi oferece um conjunto de palavras reservadas que permitem definir o comportamento das chamadas de métodos, ou seja, a ligação da chamada com a implementação desejada.

De acordo com sua forma de ligação, os métodos podem ser classificados em três tipos: *static*, *virtual* e *dynamic*. Os métodos estáticos (*static*) são o padrão do Delphi, ou seja, caso não sejam utilizadas explicitamente as diretivas *virtual* ou *dynamic*, os mesmos são considerados estáticos. A chamada desses tipos de métodos ativa a implementação

feita na classe que instanciou o objeto.

Na hierarquia de funcionários, onde estão definidos os métodos polimórficos *CalcularSalario* e *CalcularPremio*, ainda não foram aplicadas diretivas nos métodos modificando o seu tipo e, por isso, os mesmos são estáticos.

Durante a execução do sistema, variáveis podem receber objetos criados a partir de classes diferentes do tipo declarado, uma vez que estejam numa mesma hierarquia. Dessa forma a ativação dos métodos ocorre de maneira diferente entre os objetos. Isso ocorre porque para métodos estáticos, a implementação ativada é a correspondente da classe que define a variável e não da classe que define sua instância.

Tal comportamento pode ser modificado para métodos virtuais e dinâmicos, através do uso das diretivas *virtual* e *dynamic*, em conjunto com a diretiva *override*. Assim, a chamada dos métodos passa a ativar a implementação do tipo em tempo de execução, ou seja, de acordo com o tipo da instância e não o tipo definido para a variável.

O uso de *override*, garante que apenas uma implementação do método exista em tempo de execução nas classes descendentes, ocultando as implementações dos ancestrais. Porém, esses métodos devem ter exatamente a mesma assinatura, ou seja, mesmo nome do método e mesmo número, ordem e tipos dos parâmetros.

As diretivas *virtual* e *dynamic* são equivalentes, sendo diferentes no sentido de que a *virtual* otimiza a velocidade de acesso e *dynamic* otimiza o tamanho do código, sendo a *virtual* a forma mais comum e eficiente de implementar o polimorfismo. Na **Listagem 1** demonstra-se o efeito do uso dessas diretivas, através de algumas modificações na hierarquia de *Funcionario*.

Listagem 1. Definição da hierarquia de Funcionario com método virtual

```
type

  Funcionario = class
    function CalcularSalario: real; virtual;
  end;

  FuncionarioMensalista = class (Funcionario)
    function CalcularSalario: real; override;
  end;

  FuncionarioDiarista = class (Funcionario)
    function CalcularSalario: real; override;
  end;
```

Na **Listagem 1**, utiliza-se na assinatura do método *CalcularSalario* na classe *Funcionario*, a diretiva *virtual*, enquanto que nas classes descendentes utiliza-se a diretiva *override*. As implementações dos métodos continuam as mesmas.

A **Listagem 2** apresenta a utilização dos métodos com a diretiva *override*. Pressupõe-se que os valores dos campos dos objetos tenham sido modificados entre a criação dos objetos e a execução dos serviços *CalcularSalario*. **A Listagem 2** deve ser implementada em outra unit que não a *untClasses*, devendo referenciá-la.

Listagem 2. Chamada de métodos virtuais com override

```
01: var
02:   Func1: Funcionario;
03:   Func2: FuncionarioDiarista;
04: begin
05:   {Funcionario 1}
06:   Func1 := FuncionarioMensalista.Create;
07:   Func1.CalcularSalario;
08:   {Funcionario 2}
09:   Func2 := FuncionarioDiarista.Create;
10:   Func2.CalcularSalario;
11: end;
```

Durante a execução do trecho de código apresentado na **Listagem 2**, as chamadas de métodos nas linhas 07 e 10 ativam a implementação correspondente das classes *FuncionarioMensalista* e *FuncionarioDiarista* respectivamente. Embora o tipo de dados da variável *Func1* seja *Funcionario* (linha 02), o objeto armazenado nela é do tipo *FuncionarioMensalista* (linha 06). Sendo assim, em função da diretiva *override*, são executadas as implementações relativas às instâncias, e não relativas ao tipo da variável, como no caso dos métodos estáticos.

Vale a pena ressaltar que, para uso da diretiva *override*, a assinatura dos métodos deve ser exatamente igual, pois de outra forma é apresentado um erro durante a compilação. Além disso, *override* somente pode ser utilizado em conjunto com *virtual* ou *dynamic,* não podendo ser utilizado em métodos estáticos.

Para casos de polimorfismo onde a assinatura do método é diferente nos seus descendentes, podem ser utilizadas duas outras diretivas: *overload* e *reintroduce*.

A diretiva *overload* pode ser utilizada para qualquer tipo de método, seja estático, virtual ou dinâmico. Para métodos virtuais pode-se utilizar a diretiva *reintroduce* em conjunto para os descendentes. Além de permitir assinaturas diferentes de um mesmo método, a diretiva *overload* não oculta as implementações do método nos ancestrais, estando assim disponíveis mais de uma implementação simultaneamente.

Para ativar a implementação correta, são analisados os parâmetros utilizados na chamada do método, executando assim a implementação que corresponder a esses parâmetros. Porém, para os métodos virtuais, caso seja necessário, os métodos dos ancestrais podem ser ocultados, utilizando a diretiva *reintroduce.*

A **Listagem 3** apresenta um exemplo do uso das diretivas *overload* e *reintroduce*.

Nota-se a presença de uma hierarquia de classes, onde a classe *Funcionario* é ancestral direta de *FuncionarioMensalista* (linha 08). Na subclasse são alterados os métodos *CalcularSalario* e *CalcularPremio* anteriormente existentes, de forma que tenham parâmetros diferentes da classe ancestral (linhas 11 e 12). Percebe-se também, que no nível mais alto dessa hierarquia, ou seja, na classe *Funcionario*, o método *CalcularSalario* é estático (linha 05) e o método *CalcularPremio* é virtual (linha 06).

Na classe descendente são utilizadas as diretivas *overload* e *reintroduce*, onde o método *CalcularPremio* possui a diretiva *reintroduce* (linha 12), visto que é um método virtual

Listagem 3. Hierarquia com uso de overload e reintroduce

```
01: type
02:   Funcionario = class
03:     {Definições existentes}
04:     public
05:       function CalcularSalario: real;
06:       function CalcularPremio: real; virtual;
07:   end;
08:   FuncionarioMensalista = class (Funcionario)
09:     {Definições existentes}
10:     public
11:       function CalcularSalario(pHoraExtra: real): real; overload;
12:       function CalcularPremio(pTaxa: real): real; reintroduce;
13:   end;
14: implementation
15: {Implementações existentes}
16: function FuncionarioMensalista.CalcularSalario(pHoraExtra: real): real;
17: begin
18:   result := fValorMes + pHoraExtra;
19: end;
20: function FuncionarioMensalista.
      CalcularPremio(pTaxa: real): real;
21: begin
22:   result := fValorMes * pTaxa;
23: end;
```

e deseja-se que a implementação desse método no seu ancestral seja ocultada, ou seja, não disponível nessa classe nem em seus descendentes.

É necessário ressaltar que essas modificações feitas para a classe *FuncionarioMensalista* devem ser repetidas para a classe *FuncionarioDiarista*. A **Figura 1** demonstra os métodos disponíveis para a classe *FuncionarioMensalista* após a modificação das diretivas.

Na Figura anterior, observa-se que está disponível uma única implementação do método *CalcularPremio* e duas do método *CalcularSalario*. O método *CalcularSalario* tem a implementação da classe ancestral, ou seja, a classe *Funcionario*, graças à diretiva *overload*. Já o método *CalcularPremio* possui disponível apenas a implementação da própria classe *FuncionarioMensalista*, pois nesse método foi utilizada a diretiva *reintroduce*, ocultando a implementação do seu ancestral.

A diretiva *overload* pode ser utilizada também para a definição de métodos sobrecarregados dentro de uma mesma classe. Para isso, é necessário usar a diretiva na definição de todos os métodos desejados.

A diretiva *abstract* permite definir onde realmente devem ser implementados os métodos em uma hierarquia.

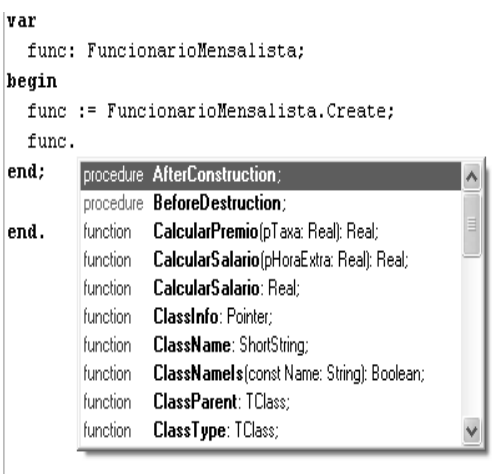


Figura 1. Métodos disponíveis utilizando reintroduce e overload

O seu uso implica que determinado método não tenha implementação na classe onde foi declarado. A implementação de fato deve ocorrer nos seus descendentes.

Ao definir pelo menos um método como abstrato, determina que a classe normalmente não vai ser instanciada e também será abstrata. Somente métodos virtuais ou dinâmicos podem ser abstratos. A **Listagem 4** demonstra o uso da diretiva *abstract*.

Listagem 4. Uso da diretiva abstract

```
01:   Funcionario = class
02:     {Definições existentes}
03:   public
04:     destructor Destroy; virtual;
05:     function CalcularSalario: real; virtual; abstract;
06:     function CalcularPremio: real; virtual; abstract;
07:   end;
08:   FuncionarioMensalista = class (Funcionario)
09:     {Definições existentes}
10:   public
11:     function CalcularSalario: real; override;
12:     function CalcularPremio: real; override;
13:   end;
14:   FuncionarioDiarista = class (Funcionario)
15:     {Definições existentes}
16:   public
17:     destructor Destroy; override;
18:     function CalcularSalario: real; override;
19:     function CalcularPremio: real; override;
20:   end;
21: implementation
22: {Implementações existentes}
23: // function Funcionario.CalcularSalario: real;
24: // begin
25: //  {Código comum aos descendentes da classe}
26: // end;
27: // function Funcionario.CalcularPremio: real;
28: // begin
29: //  {Código comum aos descendentes da classe}
30: // end;
31: function FuncionarioMensalista.CalcularSalario: real;
32: begin
33:   result := fValorMes;
34: end;
35: function FuncionarioMensalista.
      CalcularPremio: real;
36: begin
37:   result := fValorMes * 0.05;
38: end;
39: function FuncionarioDiarista.
      CalcularSalario: real;
40: begin
41:   result := fValorDia * fNumDias;
42: end;
43: function FuncionarioDiarista.
      CalcularPremio: real;
44: begin
45:   result := fValorDia * fNumDias * 0.05;
46: end;
```

Na **Listagem 4**, tem-se uma hierarquia de classes, sendo as classes *FuncionarioMensalista* e *FuncionarioDiarista*, descendentes da classe *Funcionario* (linhas 08 e 14). Nesse caso, a classe *Funcionario* possui os seus métodos *CalcularSalario* e *CalcularPremio* definidos como abstratos (linhas 05 e 06). Assim, a classe *Funcionario* torna-se uma classe abstrata, não permitindo que objetos sejam instanciados diretamente dela, obrigando aos seus descendentes a possuir implementação para tais métodos.

As implementações desses métodos nas classes *Funcionario Mensalista* e *FuncionarioDiarista* devem ser alteradas de acordo com as linhas 31 a 46 da **Listagem 4** devendo, ainda, remover tais implementações na classe *Funcionario*, pois caso contrário ocorrerá um erro de compilação, visto que métodos abstratos só possuem implementação nos

descendentes da classe (linhas 23 a 30).

O uso de *abstract* torna-se interessante, por exemplo, em casos onde se deseja realizar a chamada de um método comum a várias classes a partir do seu ancestral. Dessa forma, o ancestral define um método abstrato e os seus descendentes implementam os códigos específicos que serão executados.

A **Listagem 5** demonstra este exemplo. Para o cálculo do salário novamente se pressupõe que os valores dos campos de cada objeto tenham sido modificados logo após sua criação.

Listagem 5. Fazendo um typecasting com uma classe abstrata

```
01: var
02:   funcionarios: TList;
03:   objFunc1: FuncionarioMensalista;
04:   objFunc2: FuncionarioDiarista;
05:   i: integer;
06: begin
07:   funcionarios := TList.Create;
08:   objFunc1 := FuncionarioMensalista.Create;
09:   funcionarios.Add(objFunc1);
10:   objFunc2 := FuncionarioDiarista.Create;
11:   funcionarios.Add(objFunc2);
12:   for i := 0 to funcionarios.Count - 1 do
13:   begin
14:      Funcionario(funcionarios[i]).CalcularSalario;
15:   end;
16: end;
```

Na **Listagem 5**, nas linha 03 e 04, são declaradas variáveis que armazenam objetos de classes diferentes, porém descendentes da classe *Funcionario*. Além disso, declara-se a variável *funcionarios*, que é uma lista de objetos (*TList*), que guardará as referências dos objetos instanciados (linha 02). Na linha 07 a lista de funcionários é criada e, entre as linhas 08 e 11, são criadas instâncias de cada uma das classes e armazenadas na lista.

Após isso, a lista é percorrida calculando o salário dos funcionários nela armazenados (linhas 12 a 15). A execução do cálculo do salário ocorre na linha 14. Percebe-se que, independente de qual tipo de funcionário, ou seja, mensalista ou diarista, os mesmos podem se comportar de maneira geral como um funcionário (*typecast*), visto que herdam da classe *Funcionario*.

Dessa forma, a chamada do método abstrato *CalcularSalario*, feita à classe *Funcionario*, é redirecionada à implementação específica em cada uma das classes descendentes de acordo com o seu tipo, executando o algoritmo correto relativo à classe que instanciou o objeto. A partir do Delphi 8 existe um outro tipo de diretiva chamado *final*. Utilizar essa diretiva nos métodos de uma classe impede que os mesmos sejam redefinidos nas classes descendentes.

Associação

As associações criam conexões entre as classes, através das quais essas podem se comunicar pelo envio de mensagens. As mensagens são chamadas dos membros públicos das classes. Associações entre classes podem ser do tipo 1-para-1, 1-para-N, N-para-N.

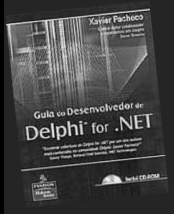
Na POO pode-se representar os dois lados de uma associação, ou apenas um deles, de acordo com o sentido do fluxo de mensagens desejado, ou seja, a navegabilidade. No contexto deste exemplo, será utilizada navegabilidade bidirecional, uma vez que não está definida a navegabilidade no modelo anteriormente apresentado.

Nesse tipo de navegabilidade qualquer dos objetos da associação possui referência ao outro objeto, ou seja, de qualquer um dos lados da associação é possível enviar mensagens ao outro.

LIVROS DE DELPHI

Estudo Dirigido de Delphi 2005
Editora Érica
R$ 76,00

Delphi 2005: Aplicações com Banco de Dados com Interbase 7.5 e MySQL 4.0.23
Editora Érica
R$ 145,00



Guia do Desenvolvedor de Delphi for .NET
Makron Books
R$ 165,00

Rave Report com Delphi
Ciência Moderna
R$ 25,00

Universidade Delphi
Digerati Books
R$ 49,90

Delphi: Progra. Banco de Dados e Web - R$69,00
Delphi 7 & E-commerce: Trein. Inter. CD - R$381,00
Delphi - R$39,00
Aplic. das Estruturas de Dados em Delphi - R$59,00
Impl. .NET no Delphi 8: Uma Visão Geral - R$31,00
CD com 50 Program./Ex. Fon. p/ Delphi - R$29,90
Delphi 8 Para Plat. .NET: Curso Completo- R$249,00
Program. em Delphi 6: Orient. por Projeto - R$56,00
Int. ao Desen. de Aplicações em Delphi - R$50,00
Conhecendo e Trabalhando com Delphi 8 - R$80,00
Estudo Dirigido de Delphi 8 - R$58,00
Pr Pascal: Ling. do Turbo Pascal do Delphi -R$52,00
Kylix: Delphi Linux com Interbase/Firebird - R$52,00
Delphi 6: Conceitos Básicos (E-Book) - R$27,90
Acessando MySQL com Delphi 7 (em CD) - R$20,00
Dominando o Delphi 7: A Bíblia - R$189,00
Redes Neurais em Delphi - R$24,00
Estudo Dirigido de Delphi 7: Avançado - R$65,00
CLX Portabilidade com Delphi 7 e Kylix 3 - R$39,00
Delphi 7: Apl. Avan. de Banco de Dados - R$87,00
Delphi 7: Conceitos Básicos - R$42,00
Estudo Dirigido de Delphi 7 - R$74,00
Programação Gráfica em Delphi 6 - R$50,00
Delphi/Kylix: Desenv. de Banco de Dados - R$72,00
Boleto Bancário em Delphi - R$35,00
Conectividade Utilizando Delphi 6 - R$40,00

FRETE GRÁTIS
Para todo o Brasil!

LivrosdeProgramacao.com.br

Para que as associações sejam estabelecidas de fato, normalmente é necessário que as classes recebam novos membros, mais especificamente campos, que correspondam ao tipo da outra classe com a qual está se associando. Tais campos podem representar uma única instância de outra classe, como também uma lista dessas, estabelecendo assim a cardinalidade desejada.

A **Listagem 6** apresenta a definição e implementação dos métodos *getDescricaoDepartamento* e *getNomeFuncionarios*, os quais permitem a troca de mensagens entre essas classes.

Listagem 6. Estabelecendo uma associação de classes 1-para-N

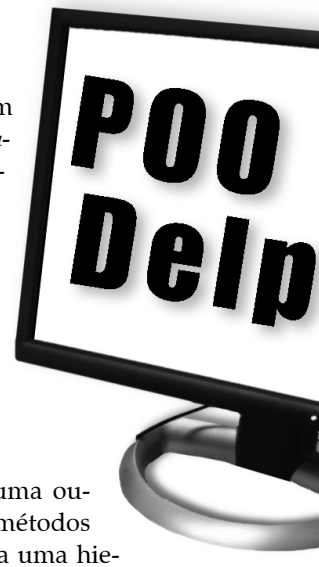
```
01: type
02:  Funcionario = class
03:  private
04:   fNome: string;
05:   fDepartamento: Departamento;
06:  public
07:   function getDescricaoDepartamento: string;
08:   {Demais definições existentes}
09:  published
10:   property Nome: string read getNome
        write setNome;
11:   property Departamento: Departamento
        read getDepartamento write setDepartamento;
12:  end;
13:  Departamento = class
14:  private
15:   fDescricao: string;
16:   fFuncionarios: TList;
17:  public
18:   function getNomeFuncionarios: TStringList;
19:   {Demais definições existentes}
20:  published
21:   property Descricao: string
        read getDescricao write setDescricao;
22:  end;
23: implementation
24:  function Funcionario.
       getDescricaoDepartamento: string;
25:  begin
26:   result := fDepartamento.Descricao;
27:  end;
28:  function Departamento.
       getNomeFuncionarios: TStringList;
29:  var
30:    i: integer;
31:  begin
32:    result := TStringList.Create;
33:    for i:= 0 to fFuncionarios.Count -1 do
34:    begin
35:      result.add(Funcionario(fFuncionarios[i]).Nome);
36:    end;
37:  end;
38: end.
```

Na **Listagem 6** pode-se perceber a presença dos métodos *getDescricaoDepartamento* e *getNomeFuncionarios*, nas linhas 07 e 18, respectivamente. Esses métodos são responsáveis pela troca de mensagens entre as classes envolvidas, solicitando informações uma da outra, visto que o acesso direto à informação não é possível, pois os campos das classes são privados.

Com a execução do método *getDescricaoDepartamento* (linhas 24 a 27) a partir de uma instância da hierarquia da classe *Funcionario*, pode-se obter a descrição do *Departamento* daquele funcionário. Já, nas linhas 28 a 37, encontra-se a implementação do método *getNomeFuncionarios* que tem como resultado uma lista com os nomes dos funcionários instanciados e armazenados por um objeto da classe *Departamento*.

Para associar um funcionário a um departamento utiliza-se o serviço *addFuncionario* da classe *Departamento*. Por outro lado, para associar um departamento a um funcionário, utiliza-se o serviço *setDepartamento* da classe *Funcionario* ou a propriedade *Departamento* da mesma classe.

As associações de classes são indispensáveis na construção de sistemas Orientados a Objetos. Normalmente essas classes são dependentes e necessitam comunicar-se através da troca de mensagens.



Interfaces

Através de *interfaces* pode-se obter uma outra forma de definir e implementar métodos comuns a várias classes, pertencentes a uma hierarquia ou não. *Interfaces* são conjuntos de definições de métodos que podem ser implementados por diferentes classes. Dessa forma, as interfaces são apenas definições de métodos, isso quer dizer que não possuem implementações, sendo as classes as responsáveis por implementar os métodos definidos na interface.

Por exemplo, a hierarquia de classes de funcionários, pode implementar uma interface para calcular seus encargos trabalhistas. Assim, essas classes passam a ser obrigadas a implementar os métodos definidos pela interface.

Na **Listagem 7**, apresenta-se a definição da interface chamada *IEncargos*, a qual pode definir diferentes métodos que tratam do cálculo de encargos. Neste exemplo apresenta-se apenas um único método chamado *ContribuicaoSindical*.

Ainda nessa listagem, apresentam-se as modificações necessárias nas classes da hierarquia de funcionários (a partir da linha 6), especificamente nas classes *Funcionario* e *FuncionarioMensalista*, para que essas passem a implementar tal interface (não se deve esquecer de declarar no *uses* da seção *interface* da *untClasses* a unit da interface criada).

Listagem 7. Usando Interfaces

```
01: type
02:   IEncargos = interface
03:   ['{41E3330E-8707-4544-87EE-B70EE7911C7A}']
04:     function ContribuicaoSindical: real;
05:   end;
06:   Funcionario = class(TInterfacedObject, IEncargos)
07:   {Definições existentes}
08:   public
09:     function ContribuicaoSindical: real; virtual; abstract;
10:   end;
11:   FuncionarioMensalista = class (Funcionario)
12:   {Definições existentes}
13:   public
14:     function ContribuicaoSindical: real; override;
15:   end;
16:   FuncionarioDiarista = class (Funcionario)
17:   {Definições existentes}
18:   public
19:     function ContribuicaoSindical: real; override;
20:   end;
```

Na **Listagem 7**, da linha 02 a 05, encontra-se a declaração da interface *IEncargos*. Por padrão do Delphi, as interfaces começam com a letra "I". Essa interface define o método

chamado *ContribuicaoSindical*, podendo ter outros métodos conforme a necessidade.

Assim como objetos descendem direta ou indiretamente da classe *TObject* do Delphi, as interfaces descendem de *IInterface*. Toda *interface* possui os métodos *QueryInterface*, *_AddRef* e *_Release*, que também devem ser implementados. Esses métodos tratam do gerenciamento e consulta dinâmica à *interface*. Para evitar essa codificação adicional pode-se determinar que classes que implementam interfaces, herdem direta ou indiretamente da classe *TInterfacedObject* do Delphi. Assim, os métodos *QueryInterface*, *_AddRef* e *_Release* já estarão implementados.

Além disso, as interfaces podem opcionalmente possuir um identificador do tipo GUID (*Globally Unique Identifier*) como mostrado na linha 03. O GUID é uma expressão em hexadecimal e pode ser gerada em tempo de design através do pressionamento das teclas Ctrl+Shift+G. Com o uso desse identificador pode-se obter referência à implementação de uma interface através de *QueryInterface*, usando por exemplo o operador *As*.

Finalmente, para determinar que uma classe implementa uma interface é necessário modificar a sua definição. Dessa forma, à frente da palavra *class* adiciona-se o nome da interface desejada, e como recomendado, estabelecendo a herança da classe *TInterfacedObject*. Tal modificação pode ser percebida na linha 06.

Vale a pena ressaltar que, essa forma de implementar interfaces no Delphi não se trata de herança múltipla. A definição da classe *Funcionario* apenas determina que essa passa a herdar da classe *TInterfacedObject* e implementar obrigatoriamente a interface *IEncargos*. Herança múltipla aconteceria se uma classe herdasse simultaneamente de duas ou mais classes, situação não permitida no Delphi. Sendo assim, uma classe pode herdar de apenas uma outra classe, mas pode implementar diversas interfaces ao mesmo tempo.

Após essas modificações, é necessário ainda adicionar à definição das classes *Funcionario, FuncionarioMensalista* e *FuncionarioDiarista* os métodos definidos pela interface, nesse caso *ContribuicaoSindical*. A assinatura dos métodos deve corresponder exatamente como definido na interface, como pode ser visto nas linhas 09, 14 e 19.

Deve-se lembrar que, devido à classe *Funcionario* implementar a interface *IEncargos*, ou ela ou todos os seus descendentes devem obrigatoriamente implementar os métodos definidos pela interface. O código da **Listagem 8** apresenta a implementação do método *ContribuicaoSindical* nas classes *FuncionarioMensalista* e *FuncionarioDiarista*.

Listagem 8. Implementação do método ContribuicaoSindical

```
01: function FuncionarioMensalista.ContribuicaoSindical: real;
02: begin
03:   result := fValorMes/30;
04: end;
05: function FuncionarioDiarista.ContribuicaoSindical: real;
06: begin
07:   result := (fValorDia * NumDias)/30;
08: end;
```

O código da **Listagem 8** retorna o valor da contribuição sindical que é correspondente, nesse caso, ao valor de um dia de trabalho em um mês de 30 dias. A chamada do método pode ser feita normalmente a partir do objeto, como qualquer outro tipo de método, visto que é implementado pela própria classe.

Porém, outras formas podem ser usadas para executar o método. A **Listagem 9** demonstra um trecho de código onde se apresentam outras duas formas de chamada do método *ContribuicaoSindical*, utilizando a interface criada. Essa listagem deve ser implementada fora da *untClasses*, em outra unit que a referencie.

Listagem 9. Chamadas do método ContribuicaoSindical

```
01: var
02:   Encargos: IEncargos;
03:   Func: FuncionarioMensalista;

04: begin
05:   {Primeira forma de chamada do método}
06:   Encargos := FuncionarioMensalista.Create;
07:   Encargos.ContribuicaoSindical;
08:   {Segunda forma de chamada do método}
09:   Func := FuncionarioMensalista.Create;
10:   (Func as IEncargos).ContribuicaoSindical;

11: end;
```

A **Listagem 9** apresenta um código correspondente para realização de chamadas de métodos implementados através de uma interface. Na primeira forma utiliza-se uma variável do tipo da interface (linha 02). Como a classe *FuncionarioMensalista* implementa a interface, tal variável pode receber a referência de um objeto dessa classe. Pode-se assim chamar o método através da interface (linhas 05 a 07).

A segunda forma pode ser realizada por meio de uma variável do tipo da classe *FuncionarioMensalista* (linha 03). Dessa forma, a variável recebe uma instância dessa classe. Posteriormente, por meio do operador *As*, a chamada é feita utilizando a interface (linhas 08 a 10).

Essa forma só é possível se a interface possuir o identificador do tipo GUID, como visto anteriormente. Caso contrário ocorrerá um erro, pois a interface utiliza o identificador para obter referência às suas implementações. Essa forma é conhecida como *interface querying*.

Estudo de Caso

A seguir, será apresentada uma aplicação simples que utiliza a maioria dos conceitos abordados, bem como o modelo anteriormente apresentado. Para isso, será adicionada uma outra classe chamada *Aplicacao* à unit *untClasses*, que manterá uma lista de funcionários e uma lista de departamentos, com o objetivo de simular a persistência (armazenamento) dos objetos.

Essa classe ainda apresenta alguns métodos que atuam nos conjuntos dos objetos. A **Listagem 10** apresenta o código da classe, com os métodos necessários para manipulação das listas.

Listagem 10. Classe Aplicacao

```
01: type
02:   Aplicacao = class
03:   private
04:     fDepartamentos: TList;
05:     fFuncionarios: TList;
06:   public
07:     constructor Create;
08:     destructor Destroy;
09:     procedure addDepartamento(pDepartamento: Departamento);
10:     procedure addFuncionario(pFuncionario: Funcionario);
11:     procedure ExcluirFuncionario(pIndice: integer);
12:     function getDescricaoDepartamentos: TStringList;
13:     function getDepartamento(pIndice: integer): Departamento;
14:     function getFuncionarios: TList;
15:     function getNomeFuncionarios: TStringList;
16:   end;
17: var
18:   aplic: Aplicacao;
19: implementation
20: constructor Aplicacao.Create;
21: begin
22:   fDepartamentos := TList.Create;
23:   fFuncionarios := TList.Create;
24: end;
25: destructor Aplicacao.Destroy;
26: begin
27:   fDepartamentos.Free;
28:   fFuncionarios.Free;
29: end;
30: procedure Aplicacao.addDepartamento(pDepartamento: Departamento);
31: begin
32:   fDepartamentos.Add(pDepartamento);
33: end;
34: procedure Aplicacao.addFuncionario(pFuncionario: Funcionario);
35: begin
36:   fFuncionarios.add(pFuncionario);
37: end;
38: procedure Aplicacao.ExcluirFuncionario(pIndice: integer);
39: begin
40:   Funcionario(fFuncionarios[pIndice]).Destroy;
41:   fFuncionarios.Delete(pIndice);
42: end;
43: function Aplicacao.getDepartamento(pIndice: integer): Departamento;
44: begin
45:   Result := Departamento(fDepartamentos[pIndice]);
46: end;
47: function Aplicacao.getDescricaoDepartamentos: TStringList;
48: var
49:   i: integer;
50: begin
51:   result := TStringList.Create;
52:   for i := 0 to fDepartamentos.Count - 1 do
53:   begin
54:     Result.Add(Departamento(fDepartamentos[i]).Descricao);
55:   end;
56: end;
57: function Aplicacao.getFuncionarios: TList;
58: begin
59:   result := fFuncionarios;
60: end;
61: function Aplicacao.getNomeFuncionarios: TStringList;
62: var
63:   i: integer;
64: begin
65:   Result := TStringList.Create;
66:   for i := 0 to fFuncionarios.Count - 1 do
67:   begin
68:     Result.Add(Funcionario(fFuncionarios[i]).Nome);
69:    end;
70: end;
```

Na **Listagem 10** temos a definição e implementação da classe *Aplicacao*. Pode-se destacar nas linhas 04 e 05 a definição das listas de funcionários e departamentos que serão mantidos por uma única instância da classe *Aplicacao*.

Os métodos *addDepartamento* (linhas 30 a 33) e *addFuncionario* (linhas 34 a 37) permitem adicionar objetos às respectivas listas, bem como os métodos *getDescricaoDepartamentos* (linhas 47 a 56) e *getNomeFuncionarios* (linhas 61 a 70) retornam listas de *strings* com as descrições de departamentos e nomes dos funcionários armazenados nas respectivas listas de objetos.

O método *ExcluirFuncionario* (linhas 38 a 42) permite a exclusão de um funcionário, além de removê-lo da lista de funcionários mantidos pela aplicação, tendo ainda o cuidado de desfazer a associação com o objeto associado da classe *Departamento*.

O método *getFuncionarios* (linhas 57 a 60) retorna a lista de objetos de funcionários que será útil para a realização dos cálculos dos funcionários armazenados pela aplicação. Já o método *getDepartamento* (linhas 43 a 46) retorna apenas um departamento de acordo com o índice do objeto armazenado na lista, sendo útil para determinar qual objeto foi selecionado a partir de um *ComboBox*, por exemplo. Por fim, os métodos *constructor* (linhas 20 a 24) e *destructor* (linhas 25 a 29) instanciam e liberam as listas da memória, respectivamente.

Um detalhe importante a respeito deste exemplo, trata-se da variável *aplic* (linha 18), que disponibiliza uma instância da classe *Aplicacao*, permitindo compartilhar as mesmas listas de funcionários e departamentos em toda a aplicação.

A **Figura 2** demonstra algumas opções do menu do formulário principal da aplicação que permitem o acesso aos demais formulários do sistema, devendo ser criado a partir de uma nova aplicação no Delphi.

As funcionalidades oferecidas pelo exemplo são cadastros dos departamentos e funcionários (no item *Cadastros* adicione "Departamentos", "Funcionários" e "Excluir Funcionários"), consultas aos funcionários de determinado departamento e calcular salários de cada funcionário. Todos esses formulários devem referenciar na sua cláusula *uses* a unit *untClasses*.

Esse formulário principal, por ser iniciado juntamente com a aplicação, tem a responsabilidade de instanciar o objeto da classe *Aplicacao*, referenciado pela variável *aplic*, definida na unit *untClasses*. A **Listagem 11** demonstra os códigos dos eventos *OnClose* e *OnCreate* desse formulário.

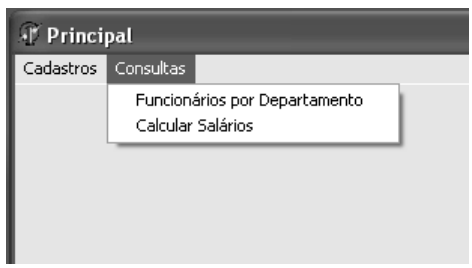


Figura 2. Formulário Principal da aplicação

Figura 3. Formulário de cadastro de Departamentos

Listagem 11. Eventos OnCreate e OnClose do formulário principal da aplicação

```
procedure TfrmPrincipal.FormCreate(Sender: TObject);
begin
  aplic := Aplicacao.Create;
end;
procedure TfrmPrincipal.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  aplic.Free;
end;
```

Na **Listagem 11** pode-se perceber a criação da instância da classe *Aplicacao* e atribuição à variável *aplic*. Ainda, nessa listagem, apresenta-se a liberação da variável da memória ao fechar o formulário, ou seja, ao finalizar a aplicação. A **Figura 3** apresenta o formulário de cadastro de departamentos, que permite a inclusão de novos departamentos na aplicação.

O cadastro de departamentos é um formulário bastante simples, com apenas um campo ("edtDescricao"), conforme definido na classe *Departamento*. A **Listagem 12** apresenta o código responsável pela inclusão de um novo departamento, implementado no botão OK.

Listagem 12. Código do botão OK do cadastro de departamentos

```
01: var
02:   depto: Departamento;
03: begin
04:   depto := Departamento.Create;
05:   depto.Descricao := edtDescricao.Text;
06:   aplic.addDepartamento(depto);
07:   edtDescricao.Clear;
08:   edtDescricao.SetFocus;
09: end;
```

O código apresentado na **Listagem 12** demonstra a criação de uma nova instância da classe *Departamento* e sua atribuição à variável local *depto* (linha 04). Após a propriedade *Descricao* do objeto, receber o texto digitado pelo usuário (linha 05), a nova instância de *Departamento* é adicionada à lista de departamentos mantida pela aplicação (linha 06).

As linhas seguintes limpam o campo em tela e posicionam o cursor para uma nova inclusão. O usuário pode incluir quantos departamentos forem necessários, sabendo-se que os mesmos estão armazenados somente em memória para efeito de exemplo.

A **Figura 4** apresenta o formulário de cadastro de funcionários, permitindo adicionar novos funcionários do tipo mensalista ou diarista, além de associá-los a um determinado departamento. Deve-se atentar para os nomes dos componentes, de forma a evitar erros de compilação (**Listagem 13**).

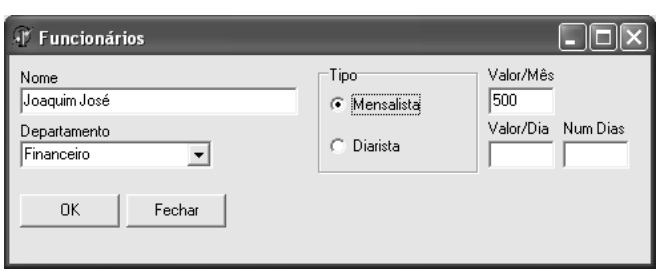


Figura 4. Formulário de cadastro de Funcionários

Nesse formulário são incluídas ainda as demais informações dos funcionários, como os valores que serão utilizados nos cálculos de salário. Além disso, associa-se um departamento ao funcionário, bem como o adiciona à lista de funcionários daquele departamento.

Embora pareça uma redundância, isso é necessário para representar a navegabilidade bi-direcional da associação. Pois, a partir de um funcionário pode-se obter seu departamento, e da mesma forma, a partir de um departamento pode-se obter a lista de seus funcionários.

A **Listagem 13** demonstra o código do formulário de cadastro de funcionários, responsável por realizar a tarefa de inclusão.

Listagem 13. Código do formulário de cadastro de funcionários

```
01: procedure TfrmFuncionario.FormActivate(Sender: TObject);
02: begin
03:   cmbxDepartamento.Items := aplic.getDescricaoDepartamentos;
04: end;
05: procedure TfrmFuncionario.btnOKClick(Sender: TObject);
06: var
07:   func: Funcionario;
08:   depto: Departamento;
09: begin
10:   case rdgrpTipoFuncionario.ItemIndex of
11:     0: begin
12:          func := FuncionarioMensalista.Create;
13:          FuncionarioMensalista(func).ValorMes :=
               StrToFloat(edtMensalistaValorMes.Text);
14:        end;
15:     1: begin
16:          func := FuncionarioDiarista.Create;
17:          FuncionarioDiarista(func).ValorDia :=
               StrToFloat(edtDiaristaValorDia.Text);
18:          FuncionarioDiarista(func).NumDias :=
               StrToInt(edtDiaristaNumDias.Text);
19:        end;
20:     else
21:       begin
22:         ShowMessage('Tipo de funcionário não selecionado');
23:       end;
24:   end;
25:   func.Nome := edtNome.Text;
26:   depto := aplic.getDepartamento(cmbxDepartamento.ItemIndex);
27:   if Assigned(depto) then
28:   begin
29:     func.Departamento := depto;
30:     depto.addFuncionario(func);
31:   end;
32:   aplic.addFuncionario(func);
33:   edtNome.Clear;
34:   cmbxDepartamento.ItemIndex := -1;
35:   edtMensalistaValorMes.Clear;
36:   edtDiaristaValorDia.Clear;
37:   edtDiaristaNumDias.Clear;
38:   edtNome.SetFocus;
39: end;
```

Na **Listagem 13** apresentam-se dois procedimentos, um para o evento *OnActivate* do formulário (linhas 01 a 04) e outro para o *click* do botão OK (linhas 5 a 39). No momento em que o formulário é criado, é necessário o preenchimento do *ComboBox* ("cmbxDepartamento") com as descrições dos departamentos anteriormente cadastrados. Para isso, é acionado o método *getDescricaoDepartamentos*, do objeto *aplic*, que retorna uma lista de *strings* contendo os nomes dos departamentos (linha 03), atribuído ao *ComboBox*.

Para o código do botão OK, são necessárias duas variáveis para trabalhar com instâncias de *Funcionario* e *Departamento*, declaradas nas linhas 07 e 08, respectivamente. Como o funcionário pode ser mensalista ou diarista, a classe que define a variável *func* é a classe ancestral *Funcionario*

(linha 07), tornando o procedimento genérico, independente do tipo do funcionário.

Após a verificação da escolha feita pelo usuário de um dos tipos de funcionário, é instanciado um objeto da classe descendente apropriada, bem como são atribuídos os valores às propriedades por meio de um *typecasting* (linhas 10 a 24).

Na linha 26, a variável *depto* recebe a referência do objeto de *Departamento*, armazenado na lista da aplicação, de acordo com o índice do item selecionado no *ComboBox*, através da execução de *getDepartamento* da classe *Aplicacao*. Na linha 27 é testada a referência do objeto retornado utilizando a função *Assigned*, que testa se a referência aponta para um objeto ou para *nil* (sem referência). De posse de uma referência válida do objeto da classe *Departamento*, é feita a associação 1-para-N com *Funcionario* de forma bidirecional. Ou seja, na linha 29, o funcionário associa-se com o departamento e, na linha 30, o departamento recebe mais um funcionário em sua lista.

A linha 32 adiciona o funcionário recém criado na lista de funcionários da aplicação e, em seguida, os campos do formulário são preparados para uma nova inclusão. Completando a manutenção dos cadastros, a **Figura 5** apresenta o formulário de exclusão de funcionários.

No formulário apresentado na **Figura 5** é possível realizar a exclusão de funcionários cadastrados na aplicação. A exclusão implica não apenas em eliminar o objeto funcionário, mas também retirar sua referência da lista mantida pelo objeto *Departamento* associado.

Além disso, deve-se retirar também sua referência da lista de objetos de funcionários mantida pela aplicação. A **Listagem 14** demonstra o código responsável pela exclusão de funcionários.

Figura 5. Formulário de exclusão de funcionários

Figura 6. Formulário de consulta de funcionários por departamento

Listagem 14. Código do formulário de exclusão de funcionários

```
01: procedure TfrmExcluirFuncionario.
      FormActivate(Sender: TObject);
02: begin
03:   lstbxFuncionarios.Items := aplic.getNomeFuncionarios;
04: end;
05: procedure TfrmExcluirFuncionario.btnExcluirClick(Sender: TObject);
06: begin
07:   if lstbxFuncionarios.ItemIndex <> -1 then
08:   begin
09:     aplic.ExcluirFuncionario(lstbxFuncionarios.ItemIndex);
10:     lstbxFuncionarios.Items := aplic.getNomeFuncionarios;
11:   end;
12: end;
```

Na **Listagem 14** estão dois eventos programados, *OnActivate* do formulário e *OnClick* do botão. No evento *OnActivate* (linhas 01 a 04) é feita a leitura dos nomes dos funcionários mantidos pela aplicação, listando-os na janela.

No evento *OnClick* (linhas 05 a 12) está o código responsável pela exclusão de um determinado objeto a partir do índice do elemento selecionado no *lstbxFuncionarios*. O método *ExcluirFuncionario* (**Listagem 10**) do objeto *aplic* é chamado para realizar a exclusão e, em seguida, é atualizada novamente a lista com os nomes dos funcionários restantes.

A seguir, a **Figura 6** apresenta o formulário de consulta de funcionários por departamento.

No formulário de consulta de funcionários por departamento, como o próprio nome indica, são listados os nomes dos funcionários que fazem parte da lista de funcionários mantida por um objeto da classe *Departamento*. A **Listagem 15** apresenta o código para o funcionamento desse formulário. Deve-se atentar para os nomes dos componentes, de forma a evitar erros de compilação.

Listagem 15. Código do formulário de consulta de funcionários por departamento

```
01: procedure TfrmFuncionariosPorDepartamento.
      FormActivate(Sender: TObject);
02: begin
03:   lstbxFuncionarios.Clear;
04:   cmbxDepartamento.Items := aplic.getDescricaoDepartamentos;
05: end;
06: procedure TfrmFuncionariosPorDepartamento.
      cmbxDepartamentoChange(Sender: TObject);
07: var
08:   deptoConsulta: Departamento;
09: begin
10:   lstbxFuncionarios.Clear;
11:   deptoConsulta := aplic.getDepartamento(
        cmbxDepartamento.ItemIndex);
12:   lstbxFuncionarios.Items := deptoConsulta.getNomeFuncionarios;
13: end;
```

Primeiramente, de maneira idêntica ao cadastro de funcionários, o *ComboBox* de departamentos ("cmbxDepartamento") é preenchido com as descrições dos departamentos mantidos na lista da aplicação (linhas 01 a 05). Posteriormente, ao selecionar um departamento, o evento *OnChange* do *ComboBox* é acionado (linhas 06 a 13).

Com isso, a variável *deptoConsulta*, do tipo *Departamento*, recebe a referência ao departamento selecionado através do índice do *ComboBox* (linha 11). Por último, o método *getNomeFuncionarios* do objeto *Departamento* é acionado, retornando uma lista com nomes dos funcionários que estão associados àquele departamento. Essa lista é atribuída ao *ListBox* ("lstbxFuncionarios") para apresentação (linha 12).

Por fim, a **Figura 7** apresenta o formulário de cálculos de salários através da lista de todos os funcionários mantidos pela aplicação com seus referidos cálculos, independente do departamento.

Através da **Figura 7** pode-se ver os nomes dos funcionários, seu tipo (nome da classe a partir da qual foi instanciado) e descrição do departamento em que está lotado, seguido dos resultados dos cálculos de salários, prêmios e contribuições sindicais, respectivamente, calculados em função da classe que instanciou o objeto.

A **Listagem 16** apresenta o código para geração desses resultados. Deve-se novamente atentar para os nomes dos componentes, de forma a evitar erros de compilação.

Listagem 16. Códigos da tela de cálculo de salários

```
01: procedure TfrmCalcularSalarios.FormActivate(Sender: TObject);
02: var
03:   funcionarios: TList;
04:   i: integer;
05: begin
06:   lstbxSalarios.Clear;
07:   funcionarios := aplic.getFuncionarios;
08:   for i := 0 to funcionarios.Count - 1 do
09:   begin
10:     lstbxSalarios.Items.add(Funcionario(
          funcionarios[i]).Nome + ' - ' +
11:       Funcionario(funcionarios[i]).ClassName +
12:       ' - ' + Funcionario(funcionarios[i]).
          getDescricaoDepartamento + ' - ' +
13:       FormatFloat('##0.00', Funcionario(
          funcionarios[i]).CalcularSalario) + ' - ' +
14:       FormatFloat('##0.00', Funcionario(
          funcionarios[i]).CalcularPremio) + ' - ' +
15:       FormatFloat('##0.00', Funcionario(
          funcionarios[i]).ContribuicaoSindical));
16:   end;
17: end;
```

Na **Listagem 16** apresenta-se o código correspondente ao evento *OnActivate* do formulário de cálculo de salários. Esse código é responsável por acionar os cálculos e apresentar as informações de todos os funcionários mantidos pela aplicação. Para isso utiliza-se de uma variável do tipo *TList* para referenciar a lista dos funcionários (linha 07), obtida a partir do objeto *aplic* que representa a aplicação.

Essa lista é percorrida para que cada funcionário seja processado e seus dados sejam adicionados ao *ListBox* ("lstbxSalarios"). Na linha 10 é recuperada a informação do nome do funcionário através da propriedade *Nome* da classe *Funcionario*.

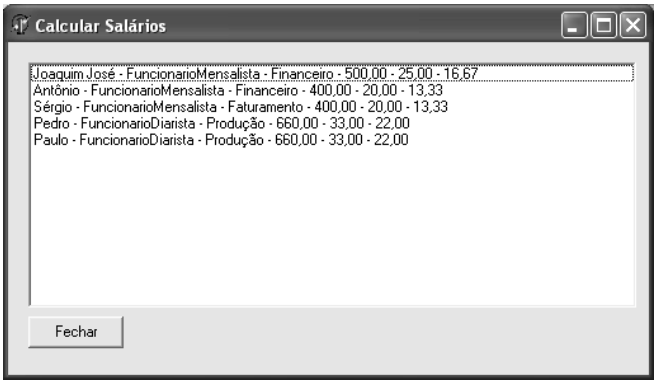


Figura 7. Formulário de consulta aos cálculos de salários de funcionários

Utiliza-se o método *ClassName* para obter o nome da classe que define o objeto (linha 11). Recupera-se a descrição do departamento associado ao funcionário, por meio do método *getDescricaoDepartamento* da classe *Departamento* (linha 12). São calculados e recuperados os valores de salário, prêmio e contribuição sindical de cada funcionário (linhas 13 a 15). Percebe-se nessa listagem que é utilizada apenas a classe ancestral *Funcionario* para fazer referências aos objetos, não sendo necessário referenciar cada tipo de funcionário específico.

Essa é uma característica marcante de programação Orientada a Objetos. Exceto pelo momento da instanciação, não é preciso verificar o tipo de um objeto numa hierarquia, pois esse é o papel das subclasses e o polimorfismo possibilita que os métodos sejam chamados corretamente em função do tipo do objeto. Com isso, economizam-se linhas de código e verifica-se um aumento significativo da legibilidade e reuso do mesmo.

Pode-se perceber ainda que o código de cada formulário é bem sucinto. Isso é uma outra característica da programação Orientada a Objetos, pois, como grande parte do código é implementado nas classes, a programação dos formulários tende a tornar-se bastante simples.

Para visualizar o funcionamento da aplicação, basta realizar a chamada dos respectivos formulários no formulário principal do projeto.

Conclusão

A programação orientada a objetos apresenta-se atualmente como uma forma eficaz para o desenvolvimento de software, e a ampla compreensão deste paradigma torna-se fundamental para que seja explorado eficientemente.

Apresentamos nestes dois artigos os conceitos mais importantes de POO, aplicando-os em um estudo de caso simples em Delphi, mas que explora a maioria destes conceitos.

Como dito anteriormente, não se teve o objetivo de ser completo em relação a esse assunto. Conceitos importantes, como persistência de objetos, não foram explorados e merecem um estudo aprofundado. ■

Ask the Expert

Perguntas e Respostas

Listar processos do usuário logado e programas minimizados para a Tray Bar

Olá Michael, gostaria de capturar os processos que estão sendo executados no perfil do usuário logado no sistema operacional. Tentei vários códigos e tudo que consegui foi listar os processos que estão rodando na máquina. Obrigada desde já.

Laira Gasparello

Via email

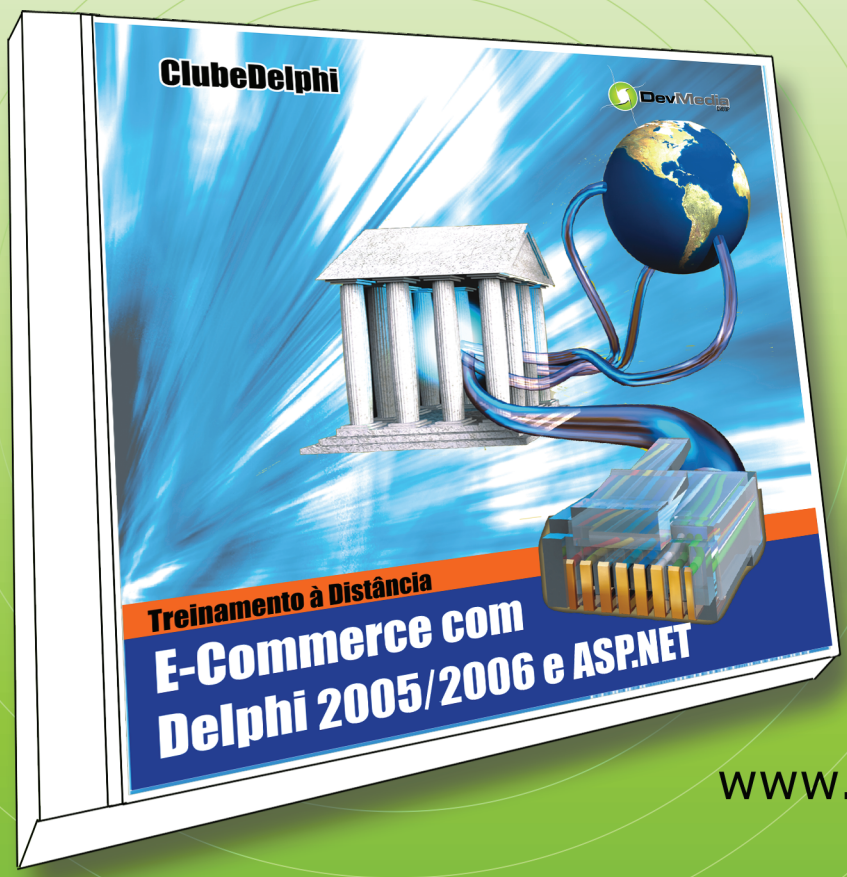
Olá Laira. Para listar os processos sendo executados no contexto do usuário logado, você pode usar o código a seguir:

```
function GetLoggedUsername: string;
var
 UserName: string;
 UserNameLen: Dword;
begin
 UserNameLen := 255;
 SetLength(userName, UserNameLen) ;
 if GetUserName(PChar(UserName), UserNameLen) then
   Result := Copy(UserName,1,UserNameLen - 1)
 else
   Result := 'Unknown';
end;

function GetUserAndDomainFromPID(ProcessId: DWORD;
  var User, Domain: string): Boolean;
var
  Token, ProcessHandle: THandle;
  Buffer: Cardinal;
  UserInfo: PTOKENUSER;
  NameUse: SID_NAME_USE;
  UserSize, DomainSize: DWORD;
  Success: Boolean;
begin
  Result := False;
  ProcessHandle := OpenProcess(PROCESS_QUERY_INFORMATION, False, ProcessId);
  try
    if (ProcessHandle <> 0) and OpenProcessToken(
      ProcessHandle, TOKEN_QUERY, Token) then
    begin
      Success := GetTokenInformation(Token, TokenUser, nil, 0, Buffer);
      UserInfo := nil;
      while (not Success) and
        (GetLastError = ERROR_INSUFFICIENT_BUFFER) do
      begin
        ReallocMem(UserInfo, Buffer);
        Success := GetTokenInformation(Token,
          TokenUser, UserInfo, Buffer, Buffer);
      end;
      CloseHandle(Token);

      if not Success then
        Exit;

      UserSize := 0;
      DomainSize := 0;
      LookupAccountSid(nil, UserInfo.User.Sid, nil,
        UserSize, nil, DomainSize, NameUse);
      if (UserSize <> 0) and (DomainSize <> 0) then
      begin
        SetLength(User, UserSize);
        SetLength(Domain, DomainSize);
        if LookupAccountSid(nil, UserInfo.User.Sid,
          PAnsiChar(User), UserSize, PAnsiChar(Domain),
          DomainSize, NameUse) then
        begin
          Result := True;
          User := StrPas(PAnsiChar(User));
          Domain := StrPas(PAnsiChar(Domain));
        end;
      end;

      if Success then
        FreeMem(UserInfo);
    end;
  finally
    CloseHandle(ProcessHandle);
  end;
end;


procedure GetLoggedUserProcesses(Processes: TStringList);
var
  ProcessSnap: THandle;
  ProcessEntry: TProcessEntry32;
  ProcessUser, LoggedUser, Domain: string;
begin
  Processes.Clear;
  ProcessSnap := CreateToolHelp32SnapShot(TH32CS_SNAPALL, 0);
  try
    if ProcessSnap = INVALID_HANDLE_VALUE then
      Exit;

    ProcessEntry.dwSize := SizeOf(ProcessEntry32);
    LoggedUser := GetLoggedUsername;

    if Process32First(ProcessSnap, ProcessEntry) then
      while Process32Next(ProcessSnap, ProcessEntry) do
        if GetUserAndDomainFromPID(
          ProcessEntry.th32ProcessID, ProcessUser,
          Domain) and SameText(ProcessUser, LoggedUser) then
          Processes.Add(ProcessEntry.szExeFile +
            ' - ' + ProcessUser + ' - ' + Domain);
  finally
    CloseHandle(ProcessSnap);
  end;
end;
```

A primeira função serve para obter o nome do usuário logado no Windows. A segunda retorna o usuário e o domínio da rede de um determinado processo, dado seu PID (Process Identification). E, por fim, a terceira lista todos os processos sendo executados na máquina e filtra aqueles que estão no contexto do usuário logado.

Treinamentos a distância.

Aprenda de forma prática, eficiente e econômica.



E-Commerce com Delphi 2005/2006 e ASP.NET

Aprenda a desenvolver poderosas aplicações Web através do Delphi 2005/2006 e as tecnologias ASP.NE WebBroker, Intraweb e WebSnap.
O curso conta com módulos textuais;
35 horas de vídeo-aulas;
fórum exclusivo para os alunos e suporte on-line.

COMPRE JÁ O SEU!

www.devmedia.com.br/curso

Outros Cursos

Delphi e Relatórios

Com este curso você desenvolverá relatórios através do Delphi 7, do Delphi 2005 e Delphi 2006, utilizando as ferramentas Rave Reports, Quick Reports e Crystal Report. Além do módulo textual o curso contém mais de 20 horas de vídeo-aulas; fórum exclusivo e suporte

Evento à distância - Introdução ao .NET

Conheça os recursos do ASP.NET com o Visual Studio 2003 e aprenda a criar páginas ASP.NET; Webservices; trabalhar com a IDE do VS; componentização; DataGrid e criar aplicações de BD com o ADO.NET! São 6h de palestras em formato de vídeo-aulas e você pode tirar suas dúvidas com os palestrantes!

Evento à distância - Novidades do Delphi 2005

Conheça os recursos do Delphi 2005 e aprenda a criar páginas ASP.NET; Webservices; trabalhar com a nova IDE do Delphi; utilizar programação OO no Delphi 2005; trabalhar em equipe com o novo Starteam e criar aplicações de BD com o ADO.NET!

DevMedia

Assinatura

msdn magazine PLUS

Mais conteúdo .NET por menos!

Sua assinatura da MSDN Magazine virou **MSDN Plus** e com isso você ganhou **470 vantagens!**

A MSDN PLUS é a combinação perfeita de conteúdo impresso, digital e áudio-visual. A partir de agora você terá acesso a um material completo de alta qualidade técnica, deixando você por dentro de tudo que acontece na plataforma .NET.

MSDN PLUS = Revista MSDN + Video Aulas + Artigos Online!

Você assinante já pode conferir este lançamento da MSDN Magazine. Com o seu login e senha você terá acesso imediato ao MSDN Plus. Hoje você já pode conferir mais de 90 vídeo-aulas e 20 artigos online. E ao longo de sua assinatura, todos os meses você terá acesso a mais 10 vídeo-aulas e 05 artigos online novos.

São cerca de 470 atualizações sobre .net durante um ano. Confira agora mais essa vantagem de ser assinante MSDN Magazine.

Por Apenas

6x de R$10,00

Saiba mais! Acesse: www.devmedia.com.br/msdn/msdn_plus.html